Engineering Heuristics and Reductions for Weighted Hypergraph b-Matching

Henrik Reinstädtler

May 3, 2023

3307518

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Jun. Prof. Dr. Felix Joos

Co-Supervisor:
Ernestine Großmann

Acknowledgments

I want to thank Prof. Schulz, Jun. Prof. Joos and Ernestine Großmann for supervising this thesis. Furthermore, I would like to thank the Studienstiftung des Deutschen Volkes for supporting my studies.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, May 3, 2023

Henrik Reinstädtler

Abstract

A hypergraph is the generalization of a graph, allowing more than two vertices to be in an edge. The weighted b-matching problem is to select the maximum weight of edges, while obeying to a capacity constraint per vertex. In this thesis, we present six novel exact reductions for this problem and introduce an iterated local search and local improvement approach. The exact reductions allow us to determine whether edges are guaranteed to be in the matching or not part of the solution. The iterated local search provides a framework of finding suitable swaps of edges, while the local improvement strategy improves the solution by exactly solving subgraphs. In experiments, we show the effectiveness of our reductions and the potential of applying iterated local search and the local improvement strategy to solutions obtained by simple weight heuristics.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Graphs are a powerful modeling tool for formulating and describing real-world problems and relationships, like social networks [7] and transportation systems [5]. One of the most prominent and well-studied problems in computer science and algorithm engineering is the matching problem in graphs. A matching is a subset of edges, that do not share a vertex. By assigning weights to the edges and asking for the highest weighted matching, a wide variety of tasks and applications can be described. This includes job assignment [70] and minimizing transportation costs [46]. Moreover, this problem can be used as a subtask for approximately solving complex problems like the metric traveling salesman problem [11]. The matching problem in graphs is in P, as it can be solved with the blossom algorithm by Edmonds [21].

In order to describe more tasks one can relax the matching constraint by introducing a capacity to the vertices, allowing them to be in a fixed number of edges in the matching. Such task is called b-matching and applications include supporting computer-aided design [41] and semi-supervised learning [45]. Additionally, in recent years non-linear, sub-modular objective functions became part of the research interest. Sub-modular functions allow modelling interesting real-life problems, like natural language processing [55] and balancing computations in quantum chemistry [25].

In addition to these changes to the functions describing the problems, a recent focus in research is the introduction of hypergraphs as a generalization of graphs. Hypergraphs allow more than to vertices to be in an edge. Therefore, non 1:1–relationships like co-authorship in scientific literature [56] or electronic circuit designs [53] can be described more accurately. The matching problem can be intuitively transferred to hypergraphs and has a variety of applications like auctioning [13] and ride-sharing [66]. The introduction of more than two vertices in an edge makes the problem more difficult to solve, as it is now reducible to the maximum weighted independent set problem and therefore in NP [44].

A natural idea is to transfer the b-matching problem to hypergraphs. A practical application that motivated our initial interest in this topic is the assignment of students to multiple classes while having multiple preferences for time slots. Students give multiple preferences for classes and time slot combinations while they are only going to attend one combination. Each class has a capacity constraint given by its room size or other factors. The edges in this example consist out of one vertex for the student and for each class they would like to attend in combination a vertex representing a time slot. This is a classical allocation problem and its solution would benefit students as they could attend the classes and time slot combinations they prefer the most.

A key idea for solving similar problems is the introduction of reductions. Therefore, we develop reductions suitable for the hypergraph b-matching problem. Furthermore, the problem sizes from real-world applications are going to rapidly grow, calling for the adaption of efficient heuristics for finding good, approximating solutions even for the matching problem in graphs.

1.2 Our Contribution

In this thesis we present six novel exact reductions, an iterated local search algorithm, a local improvement scheme and weight and degree based heuristic approaches for solving the b-matching problem in hypergraphs.

Our reductions can be categorized in two categories. Firstly, we introduce new reductions focusing on edges guaranteed to be part of the solution. Secondly, reductions that are postponing the decision and reduce the size of the hypergraph by modifying it. Lastly, we can exclude certain edges from the solution and further reduce the problem size. These reductions can help speeding up the exact solving of the hypergraph b-matching problem via integer linear programs.

Besides these reductions we introduce two improvement techniques for results obtained by simple weight heuristics. The iterated local search algorithm works by searching two edges that can replace a solution edge, yielding a better result. The local improvement scheme selects a subset of edges and solves them exactly with the help of an integer linear program.

These reductions, improvement strategies and heuristics seamlessly work together and allow us to efficiently reduce problem size and accelerate finding good solutions. By designing data structures for computing storing the results and implementing all techniques in C++, we provide a first step towards exactly and heuristically solving the b-matching problem in hypergraphs more efficiently. The reductions and heuristics developed here are also applicable to hypergraph matching and graph (b-)matching.

1.3 Structure

The remainder of this thesis is organized as follows. After defining graphs and hypergraphs, we introduce the problem in Chapter 2. In Chapter 3 we discuss recent advancements in the field. We explain the six reductions in Chapter 4 and define our framework for greedy approaches in Chapter 5. We introduce the iterated local search and a local improvement strategy in Chapter 6. Chapter 7 contains our new data structures for storing modifiable hypergraphs and b-matching. Finally, we present our experimental results for our various approaches in Chapter 8 and conclude with a discussion in Chapter 9.

CHAPTER 2

Fundamentals

In this chapter we introduce the fundamental definitions of the discussed problems and structures of this thesis. We start by introducing graphs and hypergraphs and summarize our three core problem formulations.

2.1 General Definitions

In the following we define graphs and hypergraphs as basis for our problem definitions.

Graphs. An undirected graph consists of a finite set of vertices V and a set of edges E where each edge consists of two distinct vertices. We write uv instead of $\{u, v\}$. An edge e is called *incident* to a vertex v if $v \in e$.

Two vertices u, v are *adjacent* if $uv \in E$. By $N(v) = \{u \in V \mid uv \in E\}$ we define the neighborhood of a vertex. Moreover, we define $N[v] = N(v) \cup \{v\}$ as the closed neighborhood of v.

Hypergraphs. A hypergraph $H = (V, \mathcal{E})$ is a generalization of the graph. The hypergraph H has again a finite vertex set V and an edge set $\mathcal{E}$ where $\mathcal{E} \subseteq \mathcal{P}(V)$. In the following we also consider multihypergraphs, that is $\mathcal{E}$ may be a proper multiset. Hence, we rather treat the vertices of an edge as a function $\mathcal{V}\colon \mathcal{E} \to \mathcal{P}(V)$. This allows us to have multiple edges with the same vertices, but different properties as for example weight. Furthermore, this enables us to describe our modifying reductions more accurately. An undirected graph is always a valid hypergraph. The terms *incident* and *adjacent* can be directly translated from graphs to hypergraphs: An edge e is called *incident* to a vertex v if $v \in \mathcal{V}(e)$. We define $\mathcal{E}(v) := \{e \in \mathcal{E} \mid v \in \mathcal{V}(e)\}$ as the edges that are incident to a vertex v. The degree of a vertex v is $|\mathcal{E}(v)|$. Two vertices u, v are *adjacent* if at least one edge is incident to both of them.

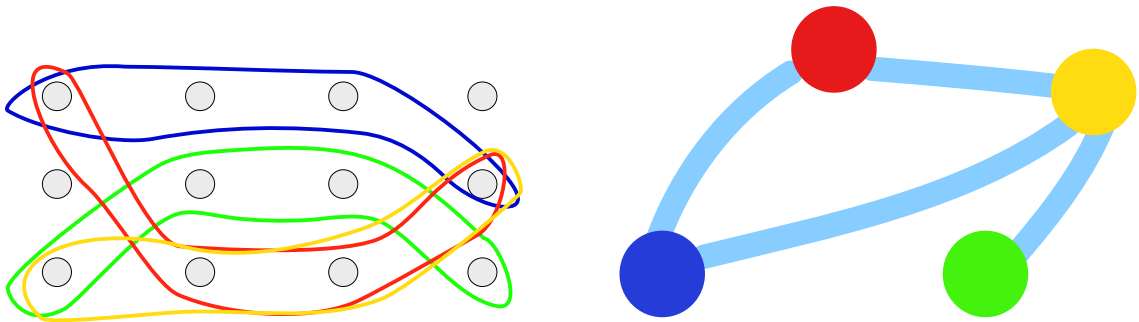


Figure 2.1: Example for a hypergraph and the corresponding line graph. Edges with a common vertex in the hypergraph are connected in the line graph.

Furthermore, two edges e, f are *adjacent* if $\mathcal{V}(e) \cap \mathcal{V}(f) \neq \emptyset$. We call two edges e, f *linked* if they are adjacent via a vertex v and $|\mathcal{E}(v)| = 2$. A set of edges S in H is *independent* if for all distinct $f, g \in S$ the vertices $\mathcal{V}(f)$ and $\mathcal{V}(g)$ are disjoint. We define $\mathcal{N}(e) := \bigcup_{v \in \mathcal{V}(e)} \mathcal{E}(v)$ as the closed neighborhood of an edge.

A *line graph* $\mathcal{L}(H)$ is the undirected graph induced by a hypergraph H neighboring information. In the line graph each edge of H is represented by a vertex and two vertices e, f of $\mathcal{L}(H)$ are adjacent if $\mathcal{V}(e) \cap \mathcal{V}(f) \neq \emptyset$, that is

$$\mathcal{L}(H) = \left(\mathcal{E}, \left\{ (e_1, e_2) \in \binom{\mathcal{E}}{2} \mid \mathcal{V}(e_1) \cap \mathcal{V}(e_2) \neq \emptyset \right\} \right). \tag{2.1}$$

In Figure 2.1 an example for a hypergraph and its line graph is shown. The line graph on the right-hand side, contains all the neighboring information of the edges of the hypergraph on the left-hand side.

Furthermore, there are special categories of hypergraphs, in particular we are interested in d-partite, d-uniform hypergraphs. A hypergraph is d-uniform if every edge contains exactly d vertices. A hypergraph is d-partite if the vertices can be partitioned in d disjoint subsets $V_1, \ldots, V_d$ and such that for every edge we have $|\mathcal{V}(e) \cap V_i| \leq 1$. An example of a 4-uniform, 4-partite hypergraph is the hypergraph in Figure 2.1.

2.2 Problem Definitions

On graphs and hypergraphs we can assign a positive weight via a mapping $c\colon V \to \mathbb{R}^+$ to vertices and to edges $w\colon E \to \mathbb{R}^+$ or $w\colon \mathcal{E} \to \mathbb{R}^+$. We can formulate different problems using this weight function and the structure of the (hyper)graph.

Maximum Weighted Independent Set. For a graph $G = (V, E)$ with a weight function $c\colon V \to \mathbb{R}^+$, we define $c(S) := \sum_{v \in S} c(v)$ for all $S \subseteq V$. A set $S \subset V$ is called *independent* if there is no edge of G with both vertices in S.

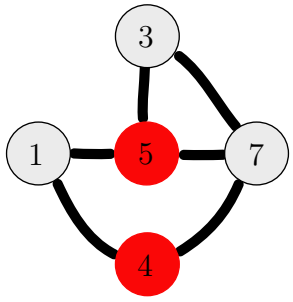


Figure 2.2: The solution for the maximum weight independent set problem illustrated in red color. The selected vertices are non-adjacent.

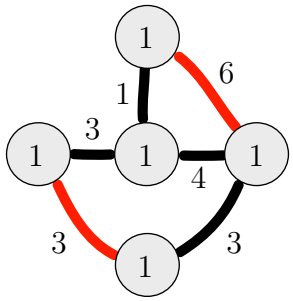


Figure 2.3: The solution for this matching problem is 9, marked in red.

The maximum weighted independent set (MWIS) problem asks for an independent set S in G that $c(S)$ is maximal. In Figure 2.2 the red vertices form an independent set with the maximum combined weight of 9. No other combination of vertices forms a higher combined weight without being adjacent.

A vertex cover is a subset of vertices C such that every edge consists of at least one vertex in the set C. The minimum weight vertex cover (MWVC) problem is to find a vertex cover C with minimal $c(C)$. The MWVC problem is the complementary problem to the MWIS problem. Thus, the weight of the MWVC is the weight of the graph without the maximum weighted independent set.

Graph Matching. For a graph $G = (V, E)$ with a weight function $w\colon E \to \mathbb{R}^+$, we define $w(M) = \sum_{e \in M} w(e)$ for all $M \subseteq E$. A subset of edges $M \subseteq E$ is called matching if all edges in M are not sharing any vertex. The maximum weighted matching (MWM) problem asks to find a matching M with maximal $w(M)$.

In Figure 2.3 an example for this is shown. The red edges do not share a vertex and have the highest possible combined weight of 9. In a *perfect* matching every vertex is adjacent to an edge that is in the matching.

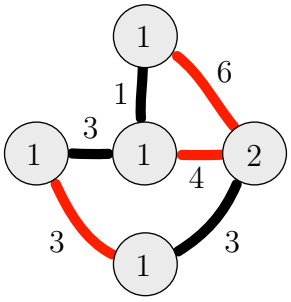


Figure 2.4: The solution for the (weighted) b-matching problem illustrated in red color. The total weight is 13 and the capacity at the right node is exhausted.

Graph b–Matching. A relaxation of the maximum weighted matching problem is the weighted b-matching problem in graphs. Given a Graph $G = (V, E)$ with a weight function $w\colon E \to \mathbb{R}^+$ and a capacity function $b\colon V \to \mathbb{N}$. For $M \subseteq E$ we define $w(M) := \sum_{e \in M} w(e)$. A set of edges M in G is called b-matching in G, if M contains at most $b(v)$ edges incident to v for every $v \in V$. The weighted b-matching problem asks to find the largest b-matching M,that is, where $w(M)$ is maximal.

Figure 2.4 shows an example for this. The red edges are the solution for the b-matching problem with a total weight of 13. Note that two edges incident to the right vertex are selected. The capacity does not have to be exhausted.

Hypergraph b-Matching. Given a hypergraph $H = (V, \mathcal{E})$ with weight function $w\colon \mathcal{E} \to \mathbb{R}^+$ and a capacity function $b\colon V \to \mathbb{N}$. For an edge set $M \subseteq \mathcal{E}$, we define $w(M) = \sum_{e \in M} w(e)$. A set of edges M in H is a b-matching in H if M contains at most $b(v)$ edges containing v for every $v \in V$. Finding the largest b-matching M in H, that is, where $w(M)$ is maximal, is called the "hypergraph b-matching problem".

The special case of $b(v) = 1$, called matching, can be reformulated as a maximum weight independent set problem of the corresponding line graph $\mathcal{L}(H)$ to a hypergraph H. Figure 2.5 shows an example for the matching problem. The solution is formed by the green and red edges with a total weight of 11. Figure 2.6 shows a b-matching with individual b values per vertex. In this configuration the blue and yellow edge form the heaviest b-matching with a weight of 13.

Cardinality Problems. All problems maximized a weight function. If the weight for all elements is equal the problems are called *cardinality* problems, because the cardinality of the set in question is maximized.

Notations. Throughout this thesis we will use the following notations: Let M be a b-matching in H. Let $M(v)$ be the edges in M that contain v and $b(v) - |M(v)|$ be the *residual capacity*. By $\mathit{blocked}(e, M) := \{v \in e \mid |M(v)| = b(v)\}$ we denote all vertices of an edge e, where the capacity is exhausted, in other words, we can not

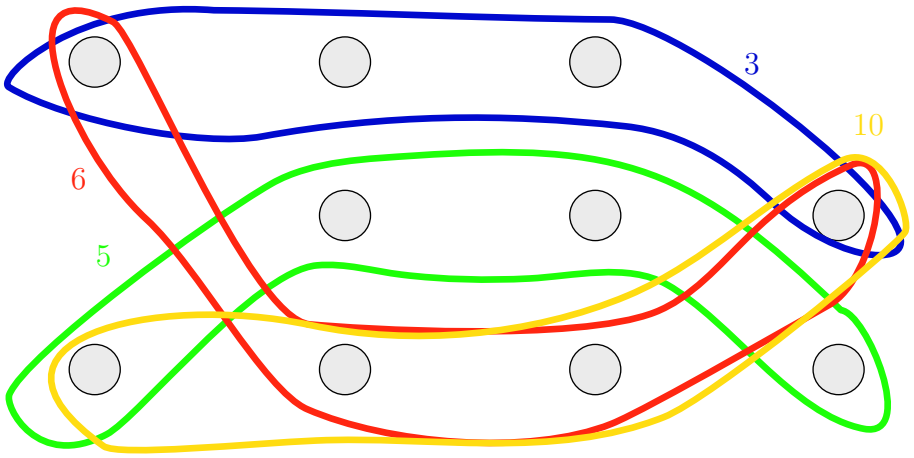


Figure 2.5: The solution for this weighted matching problem in this hypergraph is 11, formed by the red($w = 6$) and green ($w = 5$) edge.

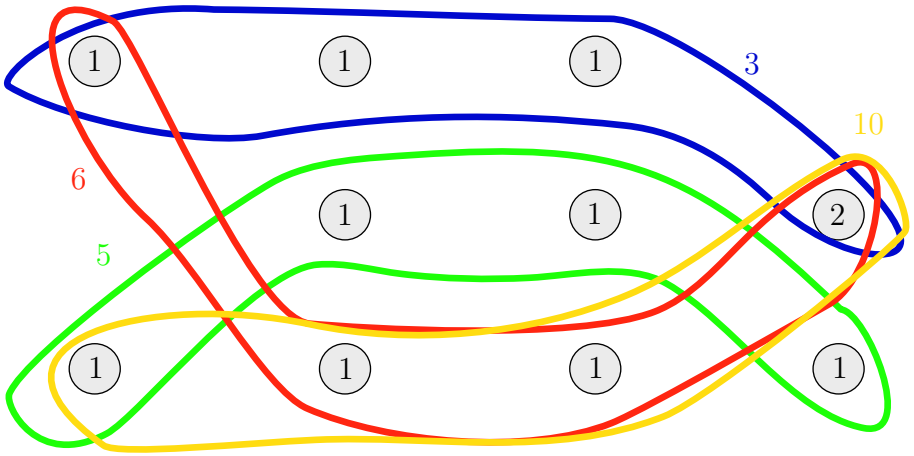


Figure 2.6: The solution for this weighted b-matching problem in this hypergraph is 13, formed by the yellow ($w = 10$) and blue edge ($w = 3$).

add further edges to the matching. An edge e with $blocked(e, M) = \emptyset$ is called *free*. Finally, we define for a finite set $X \subset \mathbb{R}^+$ $\mathrm{nmax}(X, k)$ as the k-th largest value of X as follows

$$\mathrm{nmax}(X, k) := \begin{cases} 0 & \text{if } |X| < k \\ \max X & \text{if } k = 1 \\ \mathrm{nmax}(X \backslash \{\max X\}, k-1) & \text{if } k > 1 \end{cases} . \tag{2.2}$$

CHAPTER 3

Related Work

In this chapter we discuss recent results for the maximum weighted independent set, the weighted matching and b-matching in graphs as well as in hypergraph.

3.1 Maximum Weighted Independent Set

In this section we summarize recent results for the maximum weighted independent set problem. The maximum weighted independent set problem is NP-hard according to Garey and Johnson [32]. Use cases for this problem span a variety of fields. Identifying maximum weighted independent sets can help dynamically labeling maps [69], organize long haul vehicle routing [17] and predict structural and functional sites in proteins [59]. The main focus of recent research in solving this problem are exact approaches to search the solution space, reductions to prune the search space for this problem and heuristic methods to improve solutions locally.

Exact Approaches. Branch–and–bound algorithms have dominated the field in recent years. Generally, they work by finding lower or upper bounds of an optimization problem and using this information to quickly decide, whether a current solution is feasible or not. For the MWIS setting, Warren and Hicks [71] introduced an approach based on an upper bound using clique covers. Two advancements of branch–and–bound algorithms are branch–and–reduce and branch–and–transform paradigm. Notable contributions to branch–and–reduce for this problem include the framework devised by Lamm et al. [51]. Branch–and–transform is a novel approach by Gellner et al. [34] changing the structure of the graph to by pass local optima and by adding vertices allowing more reductions to be applied.

Reductions. Most of the reductions for the MWIS problem work by adding vertices to a solution or by ruling them out of being a part of the solution. Lamm et al. [51] introduce two meta removal strategies, namely Neighbor Removal and Neighborhood Folding. Neighbor Removal works by out ruling certain vertices outright. Neighborhood Folding combines vertices and postpones decision to a later point.

Xiao et al. [72] present reductions and an execution rule set to support a branch and reduce framework. Besides sets of vertices in– or excluded in the MWIS automatically, they identify alternative and simultaneous sets, which are combination of vertices either contained mutually exclusive or at the same time.

Heuristic Approaches. There are a wide variety of heuristics used for improving solutions of the MWIS problem locally. Andrade et al. [1] developed the iterated local search technique in the cardinality setting. Local search works by finding $(1, 2)$–swaps, replacing one vertex with two feasible replacements. By iterative application and using special datastructures a local optima can be quickly found. Nogueria et al. [63] expand this work to weighted graphs and add perturbation steps to escape local optima in their HILS algorithm efficiently. For the complementary MWVC problem Li et al. [54] introduced the FastWVC heuristic focusing on initial solution quality. DynWVC1 and DynWVC2 by Cai et al. [8] is based on this work and enhance it with dynamic selection strategies, yielding different performance across different types of graphs. Gu et al. [37] use reductions in combination with a tie–breaking framework to solve the MWIS problem with little loss in optimality, while improving computation time. Exhaustively reapplying the reductions after a tie-break allows them to shrink the problem size.

Combinations of reductions, heuristic approaches, like local search and novel approaches involving graph neural networks are also feasible for the complementary MWVC problem, shown by recent results by Langedal et al. [52].

3.2 Graph Matching

The graph matching problem is a problem solvable by Edmonds [21] blossom–algorithm which is polynomial and therefore this problem lies in P. The problem serves a wide range of applications including minimizing transportation costs [46], job assignment [70] and as a subroutine in several problems, like the chinese postman tours [22] or metric traveling salesman [11].

Exact Approaches. Gabow [30] provides the fastest implementation of Edmonds blossom algorithm with a runtime of $O(mn + n^2 \log n)$. Gabow and Tarjan [31] provide an algorithm for the integer–weighted case that runs in $O(m\sqrt{n \log n} \log nN)$ for a graph with m edges, n vertices and N maximum weight.

Approximating Approaches. There are several approximation algorithms for this weighted problem. The LD (locally dominant) algorithm by Preis [67] is a linear in edge number 1/2–approximation algorithm that works by selecting locally dominant edges. Drake and Hougardy [18] designed a path–growing algorithm (PGA) that is also linear in time and works by growing paths and selecting heavier matches. Maue and Sanders [60] provide a global path growing algorithm that prioritizes heavy edges first. Birn et al. [6] presented the local max algorithm, iteratively adding local dominant edges and removing neighboring edges, which can be executed in parallel and is a $\mathcal{O}(\log^2 n)$ time, linear work algorithm. The Suitor algorithm by Manne and Halappanavar [57] is an improvement over the LD–algorithm and can be executed in parallel. For the weighted case Duan and Pettie [19] show a linear time approximation for the matching problem with an approximation guarantee provided for applications with error tolerance.

In recent years, the dynamic matching problem gained prominence. In the dynamic setting, the task is to keep the matching property, while the graph is altered. Altering the graph in the fully dynamic setting includes addition and deletion of edges [39]. Stubbs and Williams developed meta theorems for solving the weighted case and reducing it in some cases to a cardinality problem. In experimental evaluations this approach was outperformed by a random–walk based approach by Angriman et al. [3].

3.3 Graph b-Matching

The b-matching problem can be reduced to the simple matching problem according to Marsh [58] and Gabow [29], but this is infeasible on large graphs according to Khan et al. [48]. Notable applications include privacy protection through Adaptive anonymity by Choromanski et al. [10] and semi-supervised learning [45].

Exact Approaches. An overview of exact approaches can be found in Müller-Hannemann and Schwartz [62]. Most relevant, Grötschel and Holland [36] use the cutting plane technique and resort if it fails to the Padberg-Rao [64] procedure, which is a branch and cut approach. Based on belief propagation and assuming a unique solution exists Huang and Jebara [42] developed an exact algorithm for the b-matching problem.

Approximating Approaches. Mestre [61] proved that the Greedy algorithm is a 1/2–approximation and generalized the PGA algorithm by Drake and Hougardy [18] to achieve an $O(\beta m)$ time half–approximation. The LD algorithm was generalized to b–Matching by Georgiadis and Papatriantafilou [35] in a distributed fashion.

Khan et al. [48] introduced an approximation algorithm that can be executed in parallel called bSuitor, inspired by the results of Manne and Halappanavar [57] for normal matching. They report speedups of up 15 times in serial execution over a naive greedy implementation and scales well for parallel processors.

3.4 Hypergraph Matching

Hypergraph matching is a natural way to describe the process of allocating resources to machines or auctioning goods [13]. Pavone et al. [66] use an online hypergraph model to model ride-sharing opportunities in which users are allowed to leave the matching after a certain time.

The hypergraph matching problem is well studied for special classes of hypergraphs, esp. for d-uniform, d-partite hypergraphs. According to Hazan et al. [40] the maximum d-set packing problem and therefore the matching problem on d-partite, d-uniform hypergraphs can be poorly approximated within a factor of $\mathcal{O}(d/\log d)$. In general, as proven by Håstad [44] the matching problem in non-uniform hypergraphs and the maximum independent set problem are NP-hard and approximatable in $n^{1-\epsilon}$ factor.

There is a polynomial $(k+1+\epsilon)/3$-approximation algorithm for k-set packing and therefore the matching problem in d-uniform, d-partite hypergraphs proposed by Cygan [12] using local search. Furthermore, Fürer and Yu [28] improved these results with respect to the run-time. Dufosse et al. [20] introduce several heuristics to reduce the complexity of the cardinality problem by extending the well-known two Karp-Sipser [47] rules to hypergraphs. Dufosse et al. [20] present the idea of using Sinkhorn for normalisation of incident tensors as third selection rule. This idea was also formulated by Zass and Shashua [73] for normal graphs and as extension to hypergraphs.

Recently, Anneg et al. [2] showed for the non-uniform case an improved optimality bound for LP-relaxation. These results extend to b-matching.

3.5 Hypergraph b-Matching

The generalized b-matching problem is wellstudied for k-uniform hypergraphs. The b-matching cardinality problem in hypergraph has also no approximatation scheme according to El Ouali and Jäger [23], if the degree of vertices is bounded. Similiarly, El Ouali et al.[24] showed that in k-uniform hypergraphs for the cardinality problem with $2 \leq b \leq k/\log k$ there is no polynomial-time approximation within any ratio smaller than $\Omega(\frac{k}{b \log k})$. On weighted b-matching for k-uniform hypergraphs Krysta [50] gave

a greedy $k+1$ approximation, while Parekh and Pritchard [65] achieve a $(k-1+\frac{1}{k})$ approximation algorithm via linear programming. Koufogiannakis and Young [49] developed a k-approximation in a distributed fashion for weighted k-uniform hypergraphs.

CHAPTER 4

Hypergraph b-Matching Reductions

In this chapter we present our results for exact reductions for the hypergraph b-matching problem. We start with reductions based on local domination in different scenarios. In these we can directly add edges to the solution. Furthermore, we present reductions that alter the structure of the hypergraph and postpone decisions to a point after the exact solution is calculated. Finally, we present reductions that exclude certain edges from the solution and vertices from the hypergraph.

Setup. In the following, let $H = (V, \mathcal{E})$ be a hypergraph and we define M that is a subset of an optimal solution for the b-matching problem in H, obtained by applying exact reductions.

4.1 Neighborhood Removal

The first reduction concerns the domination of a neighborhood and is motivated by the optimal local solution.

Reduction 1 (Neighborhood Removal)**.** *An edge $e \in \mathcal{E}$ free can be included in M if e has a higher weight than the total sum of weights of the $(b(v) - |M(v)|)$-th largest selectable hyperedge in each of its vertices v:*

$$w(e) \geq \sum_{v \in \mathcal{V}(e)} \text{nmax}_{x \in \mathcal{E}(v), x \neq e, x \notin M}(w(x), b(v) - |M(v)|). \tag{4.1}$$

Proof. The *nmax* selects the largest weight of an edge that could be part of a matching instead of e at each vertex of the edge. This sum is the upper bound for the b-matching in the vertices of e, if e was not considered for the matching. Because the weight of the edge e is equal or higher than this sum, we can directly include e in the matching. □

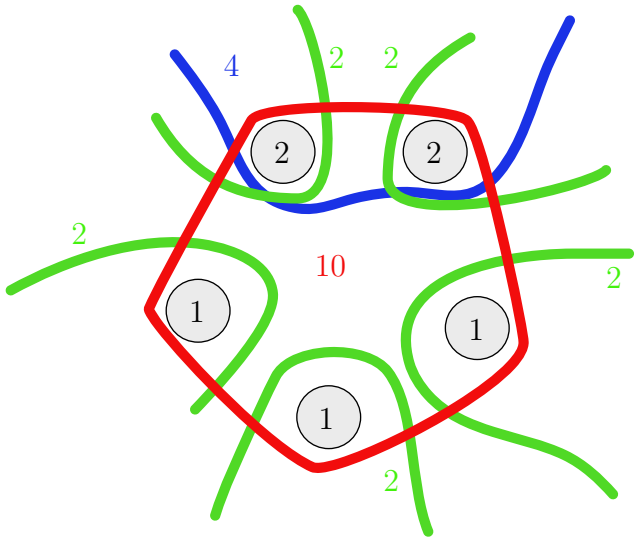


Figure 4.1: Example for the neighborhood removal reduction: The red edge dominates the sum of the $(b(v) - |M(v)|)$-th largest weights per node, in this case the green hyperedges. Therefore, it has to be part of a maximum b-matching.

An example for this can be seen in Figure 4.1. Let $M = \emptyset$, the red edge is dominating the sum of the $(b(v))$-th the heaviest edges per vertex v and is therefore known to be part of an optimal solution.

A simple algorithm for finding neighborhoods to remove is shown in Algorithm 1. We iterate over each unmatched edge and calculate the sum of weights it needs to dominate. If so, we can add it to the matching.

Note, that this bound could further be refined by requiring the weight of e to dominate the optimal solution weight in its neighborhood. Xiao et al. [72] propose a similar technique to brute–force solve neighborhoods of up to size 8 in the maximum weighted independent set setting on normal graphs.

Furthermore, we suggest to not scan all edges, but only those, where we expect a good chance of finding a domination. Either the edge size or the edge size conditional on the weight of the edge could be used as criterion. For the instances we tested on, we propose to apply this reduction only to edges up to size 4.

4.2 Weighted Isolated Edge Removal

The following reduction requires an edge to have an isolating neighborhood. In fact, we require them to form something similar to a clique in a normal graph. This reduction is inspired by a proposal by Lamm et al. [51] for the weighted independent set problem.

Algorithm 1 Algorithm for Finding Neighborhood Removal.

```
1: procedure NeighborhoodRemoval(H = (V, E), M)
2:     for e ∈ E free do
3:         w_d ← 0
4:         for v ∈ V(e) do
5:             w_d ← w_d + nmax_{x∈E(v),x≠e,x∉M}(w(x), b(v) − |M(v)|)
6:             if w_d > w(e) then
7:                 break
8:             end if
9:         end for
10:        if w_d ≤ w(e) then
11:            M ← M ∪ {e}
12:        end if
13:    end for
14: end procedure
```

Reduction 2 (Weighted Isolated Edge Removal)**.** *Suppose $e \in \mathcal{E}$ free in M with $w(e) \geq \max_{f \in \mathcal{N}(e)} w(f)$,that is $w(e) \geq w(f)$ for each f that is adjacent to e. If for all $f, g \in \mathcal{N}(e)$ exists a vertex $w \in \mathcal{V}(f) \cap \mathcal{V}(g)$ with (residual) capacity $b(w) - |M(w)| = 1$ then e is part of an optimal solution.*

Proof. Because e and all its adjacent edges contain at least one vertex with residual capacity 1, we can only select at most one edge in $\mathcal{N}(e)$ while excluding all other edges in this neighborhood. An optimal solution M^* must contain at least one edge of $\mathcal{N}(e)$ or otherwise e would be free. Given any optimal solution M^* with $f \in M^*$ for some $f \in \mathcal{N}(e) \backslash \{e\}$ then $M^* \backslash \{f\} \cup \{e\}$ is also an optimal solution if $w(e) \geq \max_{f \in \mathcal{N}(e)} w(f)$. □

An example for this is shown in Figure 4.2, where the blue edge dominates its clique neighborhood with a weight of 8. All edges share one common vertex with capacity 1 and have a lower weight than the blue edge.

In Algorithm 2 we show an algorithm to detect isolated edges. We first collect all blocking edges on capacity 1 vertices (N_b) and all other edges (N_l). If N_l is a subset of N_b and e dominates N_b weight–wise, we can directly add e to the matching.

Although this might prune the search space, it might be computational to expensive. Lamm et al. [51] use a similar technique for the maximum weighted independent set problem, they limit the search to cliques of size two or three. The structure of most hypergraphs clearly differs from that and hyperedges usually contain more vertices, which makes finding cliques harder. Therefore, we decided to limit our search in our experiments to hyperedges of size 5 or smaller and check only the candidates if there are less than 10 of them.

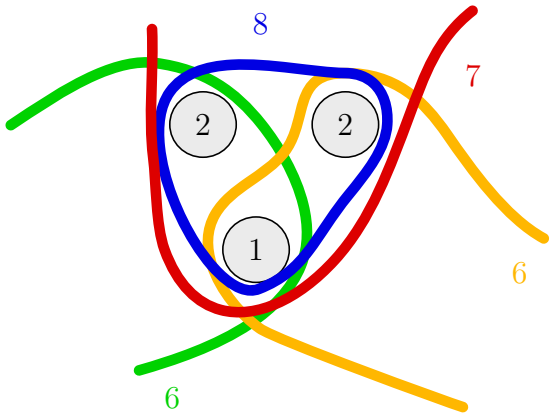


Figure 4.2: Example for the Weighted Isolated Edge Removal: All edges form a clique, as they share a common pin with capacity 1. The blue edge is guaranteed to be part of the matching, because it has the highest weight in this clique. The Neighborhood Removal is not applicable, as the blue edge does not dominate the weight of the sum.

Algorithm 2 Algorithm for Finding Isolated Edges.

```
1:  procedure IsolatedEdgeRemoval(H = (V, E), M)
2:      for e ∈ E free do
3:          N_b ← ∅
4:          N_l ← ∅
5:          for v ∈ V(e) do
6:              if b(v) − |M(v)| = 1 then
7:                  N_b ← N_b ∪ E(v)
8:              else
9:                  N_l ← N_l ∪ E(v)
10:             end if
11:         end for
12:         if max_{e_n ∈ N_b} w(e_n) ≤ w(e) ∧ N_l ⊆ N_b then
13:             M ← M ∪ {e}
14:         end if
15:     end for
16: end procedure
```

4.3 Weighted Edge Folding

The before introduced reductions work by removing vertices respectively edges from the graph. The following proposal modifies the structure of the hypergraph and postpones some decisions to a later point.

Reduction 3 (Weighted Edge Folding)**.** *Let $e \in \mathcal{E}$ be a free hyperedge and $N = \mathcal{N}(e) \backslash e$ be the edges adjacent to e. Suppose the following holds:*

(i) Each edge in N is linked to e via a vertex with residual capacity $b(v) - |M(v)| = 1$,

(ii) N is independent, that is the set of vertices for all distinct $f, g \in N$ are disjoint,

(iii) e has a higher weight than each combination without one edge alone, but smaller weight than all together, that is $w(e) > w(N) - \min_{f \in N}\{w(f)\}$ and $w(N) > w(e)$

then we can "fold" e and N in an altered hypergraph H'. The altered hypergraph H' contains a new edge e' with $w(e') := w(N) - w(e)$ and $\mathcal{V}(e') = \bigcup_{f \in N} \mathcal{V}(f)$ instead of N and e. Let M' be the optimal solution of H'. The weight of the maximum b-matching in H is $w(M') + w(e)$. If the matching M' contains e' then N is contained in an optimal solution for H. Otherwise, e is contained in a maximum matching in H.

Proof. We first show, that either e or all edges in N are contained in a maximum b-matching M^*. Requirement (ii) guarantees that N as a whole could be in M^*. Assumption (i) allows us only to have either e or any edge of N. Let $F \subset N$ be a part of the exact solution $F \subset M^*$ of the hypergraph H, we show that $F = N$ or $e \in M^*$. Because of (iii), we know that $w(e) > w(N) - \min_{f \in N}\{w(f)\} \geq w(F)$. Therefore, if F were a real subset, we could swap it for e in M^* and gain a better result proving the assumption of M^* optimal wrong. If N is not part of M^*, we can include e in the matching, because all its adjacent edges N are not part of the matching.

The vertices of e' in H' correspond to those of N in H. The vertices of e in H are only contained in e' in H', because N are all its adjacent edges, but do not have any other edge incident. Therefore, if e' is not in M' the edge e must be in an optimal solution for H and otherwise N is included in an optimal solution for H.

The formula for the weight is correct, as either e is included in the optimal matching for H, when e' is not contained in M', or the weight of M' contains $w(e')$ and thus the optimal solution in H has weight $w(M') + w(e) = w(M' \backslash \{e'\}) + w(e') + w(e) = w(M' \backslash \{e'\}) + w(N) - w(e) + w(e) = w(M' \backslash \{e'\}) + w(N)$. □

Algorithm. We present an algorithm for finding edges with two adjacent edges to fold in Algorithm 3. Only edges of size 2 are considered, we collect the neighbors on the two vertices with capacity 1 and check if they are independent. If so, we merge the independent neighboring edges and replace the neighbors and e by this merged e_n with a new weight.

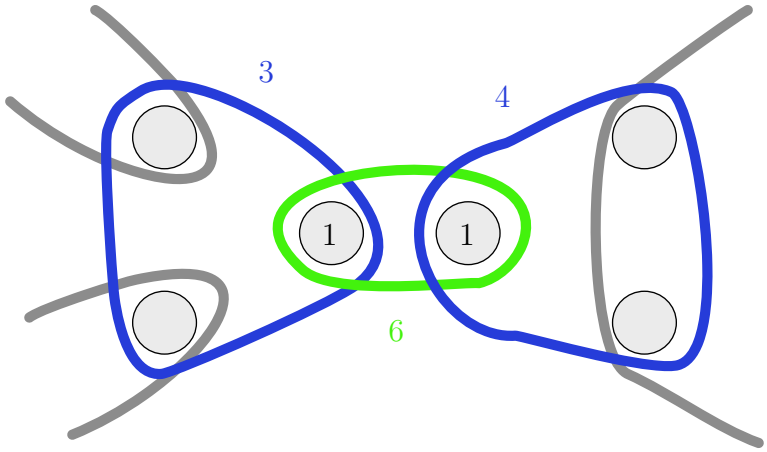


Figure 4.3: Example for the weighted vertex folding reduction: The green edge has exactly two non-adjacent neighbors (blue), that it dominates one by one, but not in total. The three edges can be folded and later be decided on.

Algorithm 3 Algorithm For Finding an Edge Folding.

procedure EDGEFOLDING($H = (V, \mathcal{E}), M$)
  for $e \in \mathcal{E}$, that is free in $M \wedge |e| = 2$ **do**
    $candidate \leftarrow True$
    $neighbors \leftarrow \emptyset$
    for $v \in \mathcal{V}(e)$ **do**
      if $|v| > 2 \vee b(v) > 1$ **then**
        $candidate \leftarrow False$
      else
        $neighbors \leftarrow neighbors \cup \mathcal{E}(v) \backslash \{e\}$
      end if
    end for
    if $candidate \wedge neighbors$ independent **then**
      if $w(neighbors) > w(e) \wedge \max_{e_n \in neighbors} w(e_n) \leq w(e)$ **then**
        $\mathcal{V}(e_n) \leftarrow \bigcup_{e \in neighbors} \mathcal{V}(e)$
        $w(e_n) \leftarrow w(neighbors) - w(e)$ ▷ Assign new weight
        $\mathcal{E} \leftarrow \mathcal{E} \backslash (neighbors \cup \{e\}) \cup \{e_n\}$ ▷ Replace edges
      end if
    end if
  end for
end procedure

4.4 Weighted Twin

In the following we present a reduction that groups non-adjacent edges with the same, independent neighborhood together. We can then decide on them, either by applying the neighborhood removal or the edge folding.

Reduction 4 (Weighted Twin)**.** *Suppose $e_1, e_2 \in \mathcal{E}$ are non-adjacent and free. Let L_i be the set of edges that are linked with e_i via a vertex with residual capacity of 1. Suppose $L_1 = L_2$, $L_i = \mathcal{N}(e_i) \backslash \{e_i\}$ and L_i independent. If $w(\{e_1, e_2\}) > w(L_1)$ then e_1 and e_2 are guaranteed to be in an optimal solution. If $w(\{e_1, e_2\}) > w(L_1) - \min_{n \in L_1} w(n)$ then the Weighted Edge Folding reduction can be applied for a combined edge of e_1 and e_2.*

Proof. Let H' be a modified version of H, replacing e_1, e_2 for an edge e' with $w(e') = w(e_1) + w(e_2)$ and $\mathcal{V}(e') = \mathcal{V}(e_1)$. Since $L_1 = L_2$ and all edges in L_2 are linked via a capacity 1 vertex we do not need to include the vertices in $\mathcal{V}(e_2)$. Any capacity constraint for an edge in L_2 at a vertex in $\mathcal{V}(e_2)$ is also present at a vertex in $\mathcal{V}(e_1)$. If $w(e') > w(L_1)$ e' is weighing more than all of its neighbors combined, thus we could always include it instead of a subset of L_1, gaining a better solution. If $w(e') > w(L_1) - \min_{n \in L_1} w(n)$ the properties for the weighted edge folding are satisfied, since the neighbors L_1 of e_1 are linked via residual capacity 1 vertex (i), L_1 is independent (ii) and the weight inequalities (iii) hold. □

An example for this folding twin reduction can be found in Figure 4.4. An algorithm for finding twins is listed in Algorithm 4. The algorithm first identifies all possible candidates that have only degree 2 vertices. Afterwards, we identify twins and either apply the neighborhood removal and add them directly to the matching. If they only dominate their neighborhood except for one edge, we merge the edges and assign a new weight to the new combined edge.

The scope of this reduction is rather small and computational expensive, because finding and validating, that two hyperedges have identical linked independent neighbors takes many comparisons. Moreover, the restriction to linked edges, required by the weighted vertex folding is constraining.

Algorithm 4 Algorithm for finding twins

```
1: procedure TwinReduction(H = (V, E), M)
2:     candidates ← ∅
3:     for e ∈ E, that is free and |e| ≤ 4 do
4:         candidate ← True
5:         neighbors ← ∅
6:         for v ∈ V(e) do
7:             if |v| > 2 ∨ b(v) − |M(v)| > 1 then
8:                 candidate ← False
9:             else
10:                neighbors ← neighbors ∪ v\{e}
11:            end if
12:        end for
13:        if candidate then
14:            candidates ← candidates ∪ {(e, neighbors)}
15:        end if
16:    end for
17:    for ∃N, e1 ≠ e2 : (e1, N), (e2, N) ∈ candidates do
18:        if N is independent then
19:            if w({e1, e2}) ≥ w(N) then          ▷ Check for weight dominance
20:                M ← M ∪ {e1, e2}
21:            else
22:                if w({e1, e2}) > w(N) − min_{e∈N} w(e) then
23:                    V(en) ← ⋃_{e∈N} V(e)
24:                    w(en) ← w(N) − w({e1, e2})          ▷ Assign new weight
25:                    E ← (E\(N ∪ {e1, e2})) ∪ {en}          ▷ Replace edges
26:                end if
27:            end if
28:        end if
29:    end for
30: end procedure
```

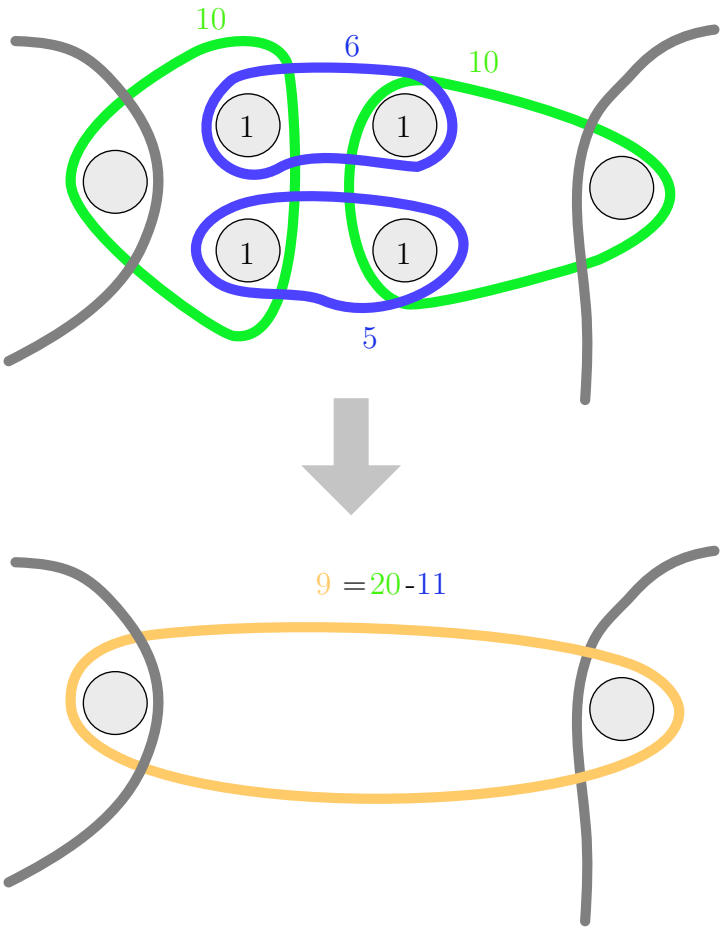


Figure 4.4: Example for the weighted twin reduction: The blue edges (u, v) are sharing the same independent linked neighbors via a vertex with residual capacity 1. As they satisfy the weight constraint for a folding ($w(\{u, v\}) > w(\mathcal{N}(u)\backslash\{u\}) - \min_{e\in\mathcal{N}(u)\backslash\{u\}} w(e)$), they can be folded, the new edge only contains the outer vertices of the neighbors and has the new adapted weight 11. If the new edge is part of the matching M' on the modified hypergraph H', the independent neighbors are part of an optimal matching, otherwise the twins are part of an optimal matching. In any case the new weight of the matching is $w(M') + w(\{u, v\})$.

4.5 Weighted Domination

The original domination proposed by Fomin et al. [26] for the maximum (unweighted) independent set reasoned, that vertices that have a bigger neighborhood can be replaced by the one having fewer neighbors. We extend this idea to edges in a hypergraph.

Reduction 5 (Weighted Domination)**.** *Let $e, f \in \mathcal{E}$ be two free edges with $w(e) \geq w(f)$. Suppose $\mathcal{V}(e)$ is a subset of $\mathcal{V}(f)$ and there is $v \in \mathcal{V}(e) \subseteq \mathcal{V}(f)$ with residual capacity of $b(v) - |M(v)| = 1$. The edge e is always an equally good choice instead of f, so f can be removed from the hypergraph.*

Proof. Since there is one vertex v with $b(v) - |M(v)| = 1$ in $\mathcal{V}(e) \subseteq \mathcal{V}(f)$ at most one of e or f can be in the maximum matching. It directly follows that e is an equal or better choice, because of its higher or equal weight and its vertices being a subset of those of f. Therefore, f can be removed from the hypergraph. □

An example for this can be seen in Figure 4.5, where the green edge is a subset of the red edge, but has a higher weight. This reduction does not work on normal, undirected graphs, because we do not look into graphs with multiple edges and the probability of having different sized edges through other reductions is very low. Therefore, we propose to limit the search for edges with different sizes.

In Algorithm 5 we show an implementation in pseudocode for finding a weighted domination. We iterate over each edge and check if it has a higher weight and strictly smaller size than its neighboring edges at a vertex with residual capacity of 1. After collecting all candidates we check, which of these candidates are strict super sets and remove them. We restrict the algorithm to strict super sets, because otherwise on normal or d-uniform hypergraphs, we would collect for the highest weighting edge at a vertex we would collect all other edges as candidates, making this approach unfavorable.

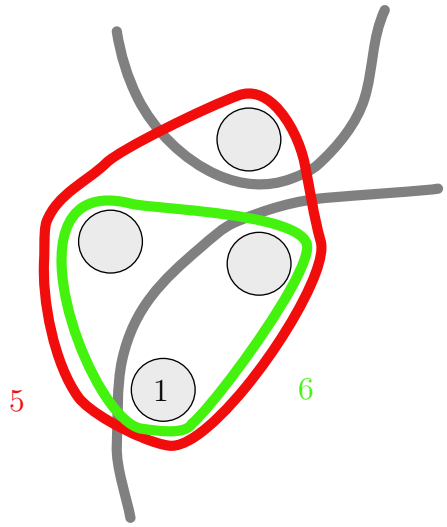


Figure 4.5: Example for the weighted domination reduction: The green edge is a subset of the red edge, has a higher weight and they share a common vertex with capacity 1.

Algorithm 5 Finding Super-set Edges, that Have a Smaller Weight.

```
1:  procedure WeightedDomination(H = (V, E), M)
2:      for e_small ∈ E do
3:          candidates ← ∅
4:          for v ∈ V(e_small) do                          ▷ Collect Candidates
5:              if b(v) − |M(v)| = 1 then
6:                  for e ∈ E(v), e ≠ e_small do
7:                      if |e| > |e_small| ∧ w(e) ≤ w(e_small) then
8:                          candidates ← candidates ∪ {e}
9:                      end if
10:                 end for
11:                 break                                  ▷ Stop collecting candidates
12:             end if
13:         end for
14:         for v ∈ V(e_small) do                          ▷ Check super-set property
15:             candidates ← {c ∈ candidates | v ∈ V(c)}
16:         end for
17:         E ← E\candidates                               ▷ Remove super sets
18:     end for
19: end procedure
```

4.6 Abundant Vertices Reduction

In addition to the reductions inspired by the maximum weighted independent set problem, we propose the removal of abundant vertices.

Reduction 6 (Abundant Vertices)**.** *A vertex $v \in V$ is abundant, if the remaining capacity $b(v) - |M(v)|$ is equal or exceeds its degree of free edges, as it does not constitute a selection problem. The vertex can be removed and edges that become empty are part of an optimal solution.*

Proof. Let m be the number of free edges containing $v \in V$ and $m \leq b(v) - |M(v)|$, all the edges incident in v could be selected at v. Thus, we can remove v from the hypergraph. If there is an edge $e \in \mathcal{E}$, only containing v, it is part of an optimal solution, since it can not be blocked at any other vertex. □

Algorithm 6 shows an implementation for this reduction. An example for this can be seen in Figure 4.6. We apply this removal strategy after every reduction, as it is not costly.

Algorithm 6 Algorithm for Removing Abundant Vertices from the Hypergraph.

```
1:  procedure ABUNDANT(H = (V, E), M)
2:      for v ∈ V do
3:          if b(v) − |M(v)| ≥ |{e ∈ E(v) | e free in M}| then
4:              for e ∈ E(v) do
5:                  if |V(e)| = 1 then          ▷ Check if edge would be empty.
6:                      M ← M ∪ {e}
7:                  end if
8:              end for
9:              Remove v from V
10:         end if
11:     end for
12: end procedure
```

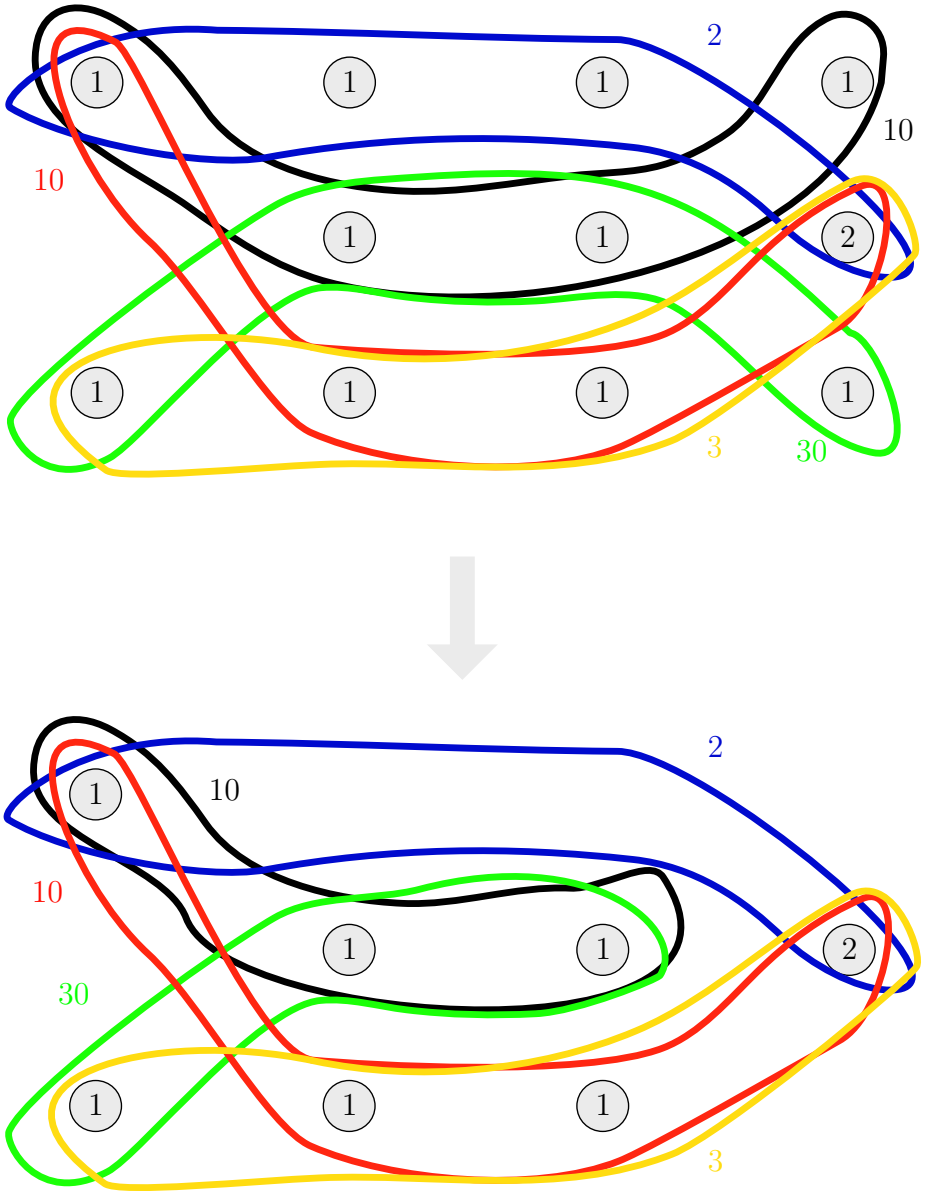


Figure 4.6: Example for the abundant vertices reduction. The hypergraph is pruned from all vertices, where no decision is to be made.

CHAPTER 5

Priority Approaches

In this chapter we discuss a naive approach for computing a matching based on priority functions. We distinguish between static priority functions and those, which dynamically update based on previous additions to the matching.

Priority Heuristics for b-Matching

We focus in this section on how to compute a good initial matching based on a priority function approach. The core idea is to assign each edge a positive priority value and then adding the highest-valuing edge greedily. The framework for selecting the edges can be seen in Algorithm 7.

Algorithm 7 A Naive Greedy Algorithm for Finding an Arbitrary b-Matching in a Hypergraph, Based on a Priority Function $h\colon \mathcal{E} \to \mathbb{R}^+$.

```
1: procedure HeuristicMatching(H = (V, E), M)
2:     while ∃e ∈ E that is free do
3:         e ← argmax_{e∈E, e free} h(e)
4:         M ← M ∪ {e}
5:     end while
6: end procedure
```

We distinguish between two different classes of priority functions. On the one hand, there are static functions, where the value only depends on the initial structure of the hypergraph and does not change during execution. On the other hand, we classify those functions, that require a recalculation conditional on the change to the hypergraph upon insertion of edges as dynamic functions.

Static Priority Functions. Intuitively, the simplest priority function for the weighted b-matching is the weight mapping itself. We can scale the weight function with the capacity at each node of the edge or the smallest capacity:

$$h_{static,ratio}(e) := w(e) \prod_{v \in \mathcal{V}(e)} b(v) \tag{5.1}$$

$$h_{static,min}(e) := w(e) \min_{v \in \mathcal{V}(e)} b(v) \tag{5.2}$$

The core idea behind these two functions is, that we want to select edges, that do not exhaust the capacity first. Note, that $h_{static,ratio}(e)$ prefers bigger edges, as each vertex of an edge contributes with its capacity, possibly bigger than 1.

Dynamic Priority Functions. We adapt the approach in Section 5.1 by accounting for only the unmatched part of the edges.

$$h_{dynamic}(e) := w(e) \prod_{v \in \mathcal{V}(e)} b(v) - |M(v)| \tag{5.3}$$

Furthermore, we propose to scale the multipliers by the degree of each vertex. The intuition is that now only vertices are influencing the product, where a decision is to be made. The multipliers are between 0 and 1, where a higher capacity leads to bigger multiplier, but not an exponential growth by edge size.

$$h_{dynamic,degree}(e) := w(e) \prod_{v \in \mathcal{V}(e)} \frac{b(v) - |M(v)|}{|\mathcal{E}(v)|} \tag{5.4}$$

All these approaches are very inflexible and greedy. Thus, they will not lead to an optimal solution in many cases. We will discuss the experimental differences in Section 8.4. We will investigate further improvements in the next chapters.

CHAPTER 6

Local Search & Local Improvement

In this chapter we present our ideas for improving results of a matching by iterated local search and local improvement strategy. Iterated local search works by finding improvements on the neighborhood scale of a problem, while a local improvement works by solving an exact solution in parts of the hypergraph.

6.1 Iterated Local Search

In this section we describe the algorithm for an iterated local search, which is inspired by the proposed one by Andrade et al. [1] and improved by Dahlum et al. [14] for the maximum independent set problem. Their algorithm works by identifying $(1, 2)$–swaps, leading to a local optima, perturbation by randomly forcing vertices into the solution and iterating this multiple times.

A $(1, 2)$–swap replaces one vertex by two vertices, that are only blocked by this first vertex. In the weighted case, the weight of these two vertices must be greater than those of the single vertex. This idea of identifying $(1, 2)$–swaps and perturbation to escape local optima can be directly transferred to the b-matching problem in hypergraphs. In the following we describe this adapted algorithm.

Hypergraph Setting. In the setting of hypergraph b-matchings edges instead of vertices are in focus. An edge is blocking another edge, if it shares vertices of the edge, where the capacity is exhausted. Therefore, we are searching for two edges in the adjacent edges, that are blocked by the first edge, but either share no common vertex or are at a non–blocking vertex.

$(1,2)$-**Swaps.** In Algorithm 8 the modified $(1,2)$-swap is shown. We first collect all neighboring edges, that satisfy the condition of being only blocked by c. In a second step we identify a pair of edges, that are not blocked at a common vertex. If we find such an edge pair we include them in the matching and remove the original edge from it.

Algorithm 8 $(1,2)$-Swaps for Weighted Hypergraph b-Matching

```
1:  procedure OneTwoSwap(H = (V, E), M)
2:      for c ∈ M do                          ▷ Every edge in the matching is a candidate
3:          l ← ∅
4:          for p ∈ V(c) do                              ▷ Go over each vertex of the edge
5:              for e ∈ E(p) do                          ▷ Only add edges to candidate set
6:                                                  that have blocked edges that are blocked by c
7:                  if e ∉ M ∧ blocked(e, M) ⊆ blocked(c, M) then
8:                      l ← l ∪ {e}
9:                  end if
10:             end for
11:         end for
12:         if |l| > 1 then
13:             if ∃x, y ∈ l : blocked(y, M\{c} ∪ {y}) ∩ blocked(x, M\{c} ∪ {x}) = ∅
                  ∧ w(x) + w(y) > w(c)  then                          ▷ Do not have
14:                                                                a common blocked vertex
15:                 M ← M \ {c} ∪ {x, y}          ▷ Replace c by x and y if applicable
16:                 M ← maximize(M)                          ▷ Restore maximal property
17:                 return true
18:             end if
19:         end if
20:     end for
21:     return false
22: end procedure
```

Perturbation. This algorithm ends up in a local optima, if executed repeatedly. Therefore, we propose to perturb the solution similarly to Andrade et al. [1]. In Algorithm 9 this perturbation framework is shown. At a first step, the number of candidates to be swapped into the solution is generated. With a low probability of $\frac{1}{2|M|}$, the number is determined by geometric distribution. Otherwise, it is simply set to 1. Then the candidates are either decided on by random or selected in the 2–neighborhood of one of $\kappa = 4$ randomly drawn edges. The distribution and κ are directly taken from the setting of the maximum weight independent set problem.

Algorithm 9 Perturbation for Weighted Hypergraph b-Matching

procedure PERTURB($H = (V, \mathcal{E}), M$)
  $\alpha \leftarrow$ random number in $[1, 2|M|]$
   $k \leftarrow 1$
  if $\alpha = 1$ **then**
    $k \leftarrow 2$
    while Coin flip is head **do** $k \leftarrow k + 1$
    end while
  end if
  if k=1 **then**
    select $x \in \mathcal{E} \setminus M$ randomly ▷ Remove all blocking edges
    $M \leftarrow (M \setminus \bigcup_{v \in blocked(x,M)} \mathcal{E}(v)) \cup \{x\}$ ▷ and replace with candidate
  else
    select $\kappa = 4$ random edges from $E \setminus M$
    $x \leftarrow$ edge that has been added/ in the solution the longest time
    $N_2 \leftarrow \bigcup_{n \in \mathcal{N}(x)} \mathcal{N}(n) \setminus \mathcal{N}(x)$ ▷ The 2-neighborhood
    $K \subset N_2, |K| \leq k$ ▷ Select up to k candidates
    for $e \in K$ **do**
      $M \leftarrow (M \setminus \bigcup_{v \in blocked(e,M)} \mathcal{E}(v)) \cup \{e\}$ ▷ Force e into solution
    end for
  end if
  $M \leftarrow$ maximize(M)
  return M
end procedure

Algorithm 10 Iterated local search for weighted hypergraph matching

procedure ILS($H = (V, \mathcal{E}), M$)
  $b \leftarrow M$
  while not stopping criterion met **do**
    $M_2 \leftarrow$ Perturb(H, M)
    while OneTwoSwap(H, M_2) **do**
    end while
    if $w(M_2) > w(M)$ **then** $M \leftarrow M_2$
    else if uniformRand$(0, 1) < \frac{1}{(w(b)-w(M_2))(w(M)-w(M_2))}$ **then** $M \leftarrow M_2$
    end if
    if $w(M) > w(b)$ **then** $b \leftarrow M$
    end if
  end while
  return b
end procedure

Iterated Local Search. The whole process is driven by the iterated local search shown in Algorithm 10. As long as the stopping criterion (in our case a time constraint) is not met, the solution gets perturbed, then improved by $(1, 2)$–swaps and finally evaluated. A better solution is always accepted as a new starting point. With a probability of

$$\frac{1}{(w(b) - w(M_2))(w(M) - w(M_2))}$$

we allow a slightly worse solution to be the starting point of our next iteration. This allows us, to faster escape a local optima. The further we are away from the optimal solution and the current starting point, the more unlikely it is, that we will change our starting point. A new global best is always accepted and stored.

Further Improvements. In order to dramatically shrink the number of iterations of Algorithm 8, we propose to use the timestamping framework, we later devise in Section 7.2 for the $(1, 2)$–swaps and after perturbation.

6.2 Local Improvement

The improvement by the iterated local search presented in the previous section is limited by finding local optima only through $(1, 2)$-swaps and perturbation. In this section we propose to find better solutions by exactly solving parts of the hypergraph, that are discovered by a breadth-first-search with a limited number of edges.

In Algorithm 11 the breadth–first search on hypergraphs is shown. We collect the edges in a list that acts as a queue and also store the already visited vertices. The loop works exactly like a classical breadth–first search, but instead of iterating over all outgoing edges, the vertices of an edge are visited and their incident edges added to the queue, if the target size has not been reached.

The local improvement scheme is displayed in Algorithm 12. It has two main parameters: t the repetition time and k the number of edges in the exactly solved part of the graph. We start the BFS t times by selecting an edge, that is not part of the solution. The idea behind this approach is similar to the local search perturbation. By starting at a non–solution edge we discover a new sub graph. The k collected edges $\mathcal{E}_e$ are then solved by an exact integer linear program that respects the residual capacity at vertices, where we were not able to collect all edges due to our size constraint n. At vertices that were not collected completely the capacity is adjusted so that edges that are part of the solution, but not collected are still forming a valid b-matching together. In this algorithm we treat the integer linear program as a black box solver for our sub problem, it can be described by

$$\max \sum_{e \in \mathcal{E}_e} x_e \cdot w(e) \quad s.t. \quad \forall v \in V \colon \sum_{e \in \mathcal{E}_e(v)} x_e \leq b(v) - |(M \backslash \mathcal{E}_e)(v)| \quad x_e \in \{0, 1\} \quad \forall e \in \mathcal{E}. \tag{6.1}$$

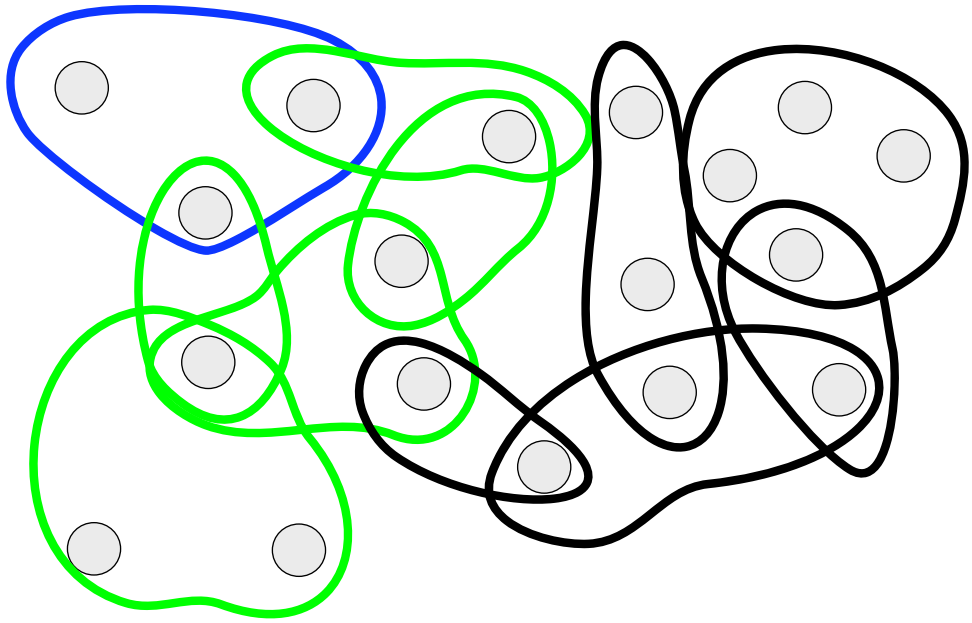


Figure 6.1: The blue edge was sampled randomly and for $k = 5$ the green edges will be solved exactly.

The maximization term is the sum of the weight of the selected edges, while the second part restricts the number of selected edges to obey to the residual capacity at each vertex. The residual capacity allows keeping edges that are not part of $\mathcal{E}_e$, but in M, in M and produces a valid b-matching.

An illustration of this can be seen in Figure 6.2. The blue edge was randomly sampled and for a collection of 5 edges, the green sub graph will be solved exactly.

Algorithm 11 Limited Breadth–First–Search for Hypergraphs

```
1: procedure BFS(H = (V, E),e,k)
2:     visited_edge ← [e]
3:     visited_vertex ← {}
4:     i ← 0
5:     while |visited_edge| < k ∧ i < |visited_edge| do
6:         i ← i + 1
7:         c ← visited_edge[i]
8:         for v ∈ V(c) do
9:             if v ∉ visited_vertex then
10:                visited_vertex ← visited_vertex ∪ {v}
11:                for d ∈ E(v) do
12:                    if d ∉ visited_edge ∧ |visited_edge| < k then
13:                        visited_edge ← append(visited_edge, d)
14:                    end if
15:                end for
16:            end if
17:        end for
18:    end while
19:    return visited_edge
20: end procedure
```

Algorithm 12 Local Improvement Algorithm for Finding Optimal Solutions in Subgraph

```
1: procedure LocalImprovement(M,H = (V, E), k,t)
2:     for i:=1,...,t do
3:         e ← random(E\M)          ▷ Sample starting edge from non–solution edges
4:         E_e ← BFS(H, e, k)       ▷ Initialize subgraph with k edges around e
5:         M_e ← ILP(E_e)           ▷ Exactly solve subgraph
6:         for f ∈ E_e do           ▷ Update solution
7:             if f ∈ M_e then
8:                 M ← M ∪ {f}
9:             else
10:                M ← M\{f}
11:            end if
12:        end for
13:    end for
14: end procedure
```

CHAPTER 7

Data Structures

In this chapter we develop a data structure to store hypergraphs and b–Matching in hypergraphs efficiently. The structure of this chapter is as follows: We first introduce the data structure to store modifiable hypergraphs and then proceed to the structure holding the matching. Finally, we present the meta information we are storing in our matching structure in order to save computational time for operations that would usually require using the hypergraph structure.

7.1 Modifiable Hypergraph

A hypergraph can be described by a vector of vectors of vertices and a list of lists of edges that contain a reference to the other list. Because the number of edges and vertices are fixed in hypergraph partioning, Schlag et al. [68] merge the list of lists into a big array storing references to start and endpoints in a separate array, making the structure more cache efficient. Some of our reductions require the hypergraph to be modifiable in that sense, that we want to merge edges and remove vertices. We do not need operations of adding edges or vertices. Therefore, we can not rely on the previous work.

A merge of multiple edges can be described by a repeated merge of two edges. We describe the merge of two edges by deactivating one edge and inserting the additional vertices of the deactivated edge into the remaining edge. Therefore, we will discuss three operations: deactivating, activating single edges, merging two edges and unmerging them.

Deactivating and Activating. For each edge and vertex we are storing which vertices or edges are incident to them. Retrieving the size of an edge or degree is a common operation. Furthermore, we would like to reduce the number of allocations needed for our operation, as resizing an array is costly. Consequently, we store the

current size of an edge or number of incident edges for a vertex separately. When we deactivate an edge, we can simply move its reference in all vertices to the current size and then reduce the stored size by one. The edge list consisting of the vertices the edge was incident is deactivated by prefixing the current size of it to negative. For activating a previously deactivated edge, we swap the edge in the vertex with the edge at position of the current size plus one and then increase the size by one. Furthermore, we can flip the sign of the current size of the edge to enable it. Thus, the number of times, when we have to allocate extra memory is limited.

Merging and Unmerging. We insert the vertices of the deactivated edge into the remaining edge. Naturally, we have to store the vertices that are newly added to the edge. Later, we unmerge an edge by removing the additional vertices of the deactivated edge and activating the edge. Furthermore, we have to check for vertices that have been deactivated in the meantime and ignore them.

7.2 b–Matching

The core idea of this data structure is to provide quick access to solution, unmatched and blocked edges. Therefore, we store the ids of the hyperedges in these three compartments. In order to access the information in which compartment a specific hyperedge is, we store the position of each hyperedge in a separate array. In Figure 7.1 an example for the first part of the data structure is shown. If we add a hyperedge to the solution, we must move all neighbors, that are becoming blocked by at least one vertex of the added edge to the blocked section. This structure also allows to quickly maximize a matching by simply inserting the first free edge until no edge is left over, a procedure outlined in Algorithm 13.

Algorithm 13 A naive greedy algorithm for finding an arbitrary b-matching in a hypergraph.

```
1: procedure MAXIMIZE(M)
2:     while ∃e ∈ E that is free do
3:         M ← M ∪ {e}
4:     end while
5: end procedure
```

Precomputed Information. This basic structure allows us to implement most of our algorithms. Nevertheless, there are basic operations that are still costly. In the following we give an intuition, why we need to store additional information. This includes information about the number of blocked vertices, matched edges and blocked edges at vertices and latest change information on the edges.

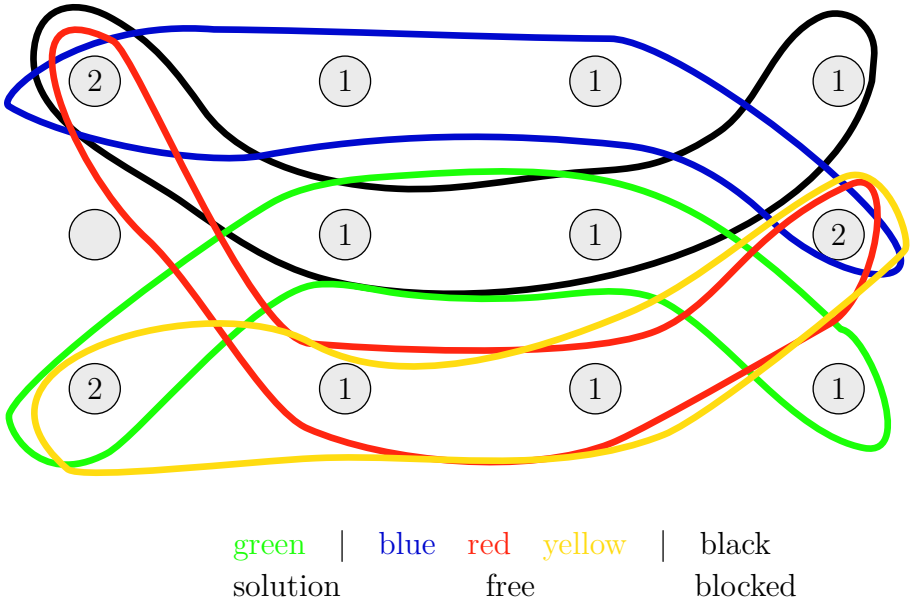


Figure 7.1: Data structure for storing a b-matching. We store the indices of the edges in three sections. The green edge is part of the solution at this stage. It blocks the black edge. The yellow edge is not blocked, because the capacity at their common vertex is 2.

Number of Blocking Vertices. The number of blocking vertices in an edge is a useful shortcut. Otherwise, checking if an edge is still blocked after the removal of a neighboring edge requires searching in all its vertices for blocking incident edge. With this information we only need one pass over the vertices of the edge we are removing and through updating the information on the incident edges. We can decide if they have to be moved in the data structure to a different section. For the matched edges we store, how many vertices they block. This allows us to quickly identify, whether a blocked edge is only blocked by one edge and could be swapped for it.

Matched and Blocked Edges at a Vertex. In two separate arrays we store how many edges are in the matching or blocked at a vertex. This is needed, as we often access this information, and we do not want to recompute this every time by iteration. The information, how many edges are not blocked is not that interesting and can be simply computed from the node degree and the blocked or matching count.

Timestamping Changes in the Neighborhood. As we do not want to iterate over all edges after a change, we need an additional array storing a "timestamp". We update the timestamp every time we add or remove a neighbor to the solution or (in case of a reduction) change the hypergraph in the neighborhood. This allows to quickly detect changes during the iteration by simply comparing the timestamp to the previous one.

CHAPTER 8

Experimental Evaluation

In this chapter we are going to discuss the results obtained by our algorithms devised in Chapter 4 and 6. We implemented them and the data structures defined in Chapter 7 using C++. The KaHyPar library by Schlag et al. [68] influenced our interfaces of the several classes. In order to solve integer linear programs and obtain exact results we use Gurobi [38]. For comparison, we are linking the libraries by Khan et al. [48] for b-matching on normal graphs and the library for d-uniform, d-partite hypergraphs by Dufosse et al. [20] into our benchmark suite. They work with separate input file types as specified by the respective authors. The structure of this chapter is as follows, we first introduce the setup for the experiments, our benchmark instances and then present the results for our experiments.

8.1 Methodology

In this section we describe our methodology for our experiments including the system we used, how we executed experiments and how we compare them graphically.

System. All of our experiments were carried out either on a 16–core Intel(R) Xeon(R) Silver 4216 running at 2.10 GHz equipped with 96 GB of RAM. All programs where compiled with `clang++-15` and the following flags: `-O3 -march=native`.

Execution Policies. For experiments with our heuristics that incorporate randomness such as iterated local searches or ILPs solved by Gurobi, we run the experiments 10 times and take the arithmetic mean as result per instance. Deterministic experiments were only executed once iff the results (size, weight) and not the time was measured. The experiments were scheduled in parallel up to the numbers of cores of the machine and the number of cores used by Gurobi was limited to one.

Instance	Edges	Nodes
mouse-gene	14 461 095	45 101
Fault_639	13 987 881	638 802
astro-ph	121 251	16 706
cond-mat-2005	175 693	40 421
gas_sensor	818 224	66 917
Reuters911	148 038	13 332
turon_m	778 531	189 924
kron_g500-logn21*	91 040 932	2 097 152
dielFil.V3real	44 101 598	1 102 824
bone010	35 339 811	986 703

Table 8.1: Instances from the Florida Sparse Matrix Collection by Davis and Hu [15], selected by Khan et al. [48]. Due to its large size we exclude **kron_g500-logn21** in some of experiments with Gurobi.

Performance Profiles. For comparison we are using performance profiles as proposed by Dolan and Moré [16]. We plot which fraction of instances is solved by an algorithm to size of at least $\tau w(M_{opt})$, $0 < \tau \leq 1$ and M_{opt} being the best matching reported by all heuristics. Thus, having a fraction near 1.0 for a high τ is considered a good performance, because a high fraction of instances is then solved to near optimum.

Time Performance Profiles. Similarly, we are using these profiles by Dolan and Moré [16] to compare execution duration of approaches. Here τ is greater than 1, for each instance the time is marked relative as multiplicative of the minimum time needed to solve the instance exactly.

8.2 Instances

We test our algorithms on a variety of hypergraph and undirected graph types.

8.2.1 Graphs

The graphs used span a wide variety of purposes. We rely on two types of graphs, real–world instances and synthetically generated graphs, mimicking social networks.

Florida Sparse Matrix Collection. We use the graphs from the Florida Sparse Matrix Collection by Davis and Hu [15], as proposed by Khan et al. [48]. The list of the used instances are shown in Table 8.2.1.

R–MAT. We use KaGen by Funke et al. [27] to generate a total of 90 instances of the R-MAT type using the generator by Hübschle-Schneider and Peter Sanders [43]. RMAT graphs have a recursive structure allowing them to mimic social networks according to Chakrabarti et al. [9]. We generate 30 instances with 2^{16},2^{17} and 2^{18} nodes and resp. 265 860, 2 658 600 and 26 586 000 edges. As parameter set for the recursive structure of 30 graphs (ten per size) each we chose $(0.55, 0.15, 0.15, 0.15)$, $(0.45, 0.15, 0.25, 0.15)$ and $(0.25, 0.25, 0.25, 0.25)$, these parameters are those selected by Khan et al. [48].

8.2.2 Hypergraphs

We focus on three types of hypergraphs: d-uniform, d-partite hypergraphs, hypergraphs derived from incident matrices and those derived from neighborhoods of undirected graphs.

d-uniform, d-partite Hypergraphs. As special class of hypergraphs we use d-uniform, d-partite hypergraphs, proposed by Dufosse et al. [20]. In these hypergraphs edges contain each one vertices out of d categories and all have the size of d. For $d = 6$ 160 hypergraphs were generated by the planted scheme. The planted hypergraphs by Dufosse et al. [20] are relatively sparse.

Hypergraphs Derived from Incident Matrices. Hypergraphs can be retrieved from incident matrices. A non-zero entry at index (x, y) defines a connection between edge x and vertex y. We collected 354 matrices from the suite sparse collection categories combinatorial and circuits by Davis and Hu [15]. The instances contain between 6 and 564 480 vertices and 10 to 376 320 hyperedges. In most of the experiments we focus on 329 instances with up to 20 000 hyperedges and refer to these instances as Matrix Market collection.

Hypergraphs Derived from Neighborhood Structure of Graphs. Furthermore, we derived hypergraphs from graphs by their neighborhood structure. For each vertex in the original graph we generate a hyperedge with its neighbors. We selected the citation and co-authorship networks, as used by Geisberger et al. [33] and available via Bader et al. [4]. The transformation is very similar to those derived from incident matrices: The graph can be represented by the adjacency matrix (with a full diagonal), which then can be used as incident matrix for the transformation. We collected 5 instances with between $227,320$ and $540,486$ hypervertices and hyperedges.

8.2.3 Weights

Lastly, we have to define, how we assigned weight to the edges and vertices in our hypergraph and graph instances.

Reduction	Constraint	Default
Neighborhood Removal	edge size	4
Isolated Edge Removal	edge size	5
Isolated Edge Removal	clique size	10
Weighted Domination	edge size	6
Twin Folding	edge size	4
All	iterations	10

Table 8.2: Overview over the constraints of the reductions.

Edge Weights. We assign each edge a uniformly distributed random weight between 1 and 100. For some experiments we used uniform edge weights.

Vertex Weights. In our experiments we will test uniform capacities between 1 and 10, where we assign all vertices the same capacity. The implementation is also able to handle non-uniform capacities.

The remainder of this chapter is organized as follows: We first investigate how our reductions speed up the black box solver by Gurobi [38] on hypergraphs. Then we compare our priority functions with bSuitor by Khan et al. [48] on normal graphs. We compare our iterated local search with Karp–Sipser scaling by Dufosse et al. [20] on d-uniform, d-partite hypergraphs with uniform edge weight. Finally, we benchmark our iterated local search with the local improvement scheme we devised in Section 6.2.

8.3 Reductions and Speedup

In this experiment we investigate, how well our reductions can speed up solving b-matching problems with Gurobi [38] as black–box solver. We apply our search algorithms for the reductions in the constrained setting up to ten times and pass the resulting core problem to Gurobi and compare its total run time to the time it takes to solve the whole (hyper-)graphs without reduction. Furthermore, we test our reductions with and without constraint, the constraints are listed in Table 8.3. The runtime of the program with reductions includes the search time to find the reductions. We set a timeout for the Gurobi computations of 3 600 seconds and test it on uniform capacities of 1, 3 and 5. Only instances that Gurobi can solve exactly in this time limit are considered. In the figures the exact solver without reductions is referred to by ilp_exact, the one with reductions with the constraints by reductions_ilp and the one without constraints by reductions_unconstrained_ilp.

6–**uniform,** 6–**partite Hypergraphs.** In Figure 8.1 we show a time performance profile on the planted hypergraphs, generated by the scheme provided by Dufosse et al. [20]. For the 34 instances that Gurobi can solve with capacity 1 we can report a geometric mean speed up of roughly 25%. About 20% of the instances are not speed up by applying the reductions. The 40 instances on capacity 3 are speed up by a factor of 5.9 and the 80 instances with capacity 5 are speed up by a factor of 5. For these capacities every instance is speed up by applying reductions. The unconstrained reduction version is always slower, for capacity 3 and 5 roughly 4 to 5 times for the slowest instance.

Table 8.3 shows the impact of the reductions on the hypergraph instances. The higher the capacity the more edges are removed by our reductions. The unconstrained version sometimes removes more edges and vertices, but most of the time the average speedup relative to solving it directly is smaller than in the constrained version. For instances with capacity 5 only a few edges are passed on to Gurobi, explaining the speedup.

Matrix Market Hypergraphs. The results on the hypergraphs based on incident structure from the Florida Sparse Matrix collection are shown in Figure 8.2. For capacity 1 we can report a speed up of about 20%, for capacity 3 the run time only improves slightly and for capacity 5 we can half the execution time on average for the restricted reductions. For all capacities there are few instances that do not benefit from the restricted reductions. The unrestricted reductions take on some, but few instances over 1000 times the optimal time.

Neighborhood Structure Hypergraphs. In Figure 8.3 the results for the hypergraphs based on neighboring structure from citation networks are shown. For the 5 instances of hypergraphs based on the neighborhood structure of graphs we can report a speed up of nearly factor two for capacity 1, 10% for capacity 3 and 15% for capacity 5. For capacity 5 only four of the five instances can be solved exactly during the time given. The unconstrained version is always slower than the constrained version of the reductions, but faster than the version without any reductions. The overall speed up is not that pronounced as in the case of the planted hypergraphs.

Florida Sparse Matrix Collection Graphs. In Figure 8.4 the comparison between the restricted version of the reductions and the solving only with Gurobi [38], we omit the unconstrained version of reductions, since no reduction constraint is applicable to these graphs. The six out of nine solvable instances roughly take the same time for the reductions and the direct solving. For capacity 3 there is one instance that can be solved five times faster.

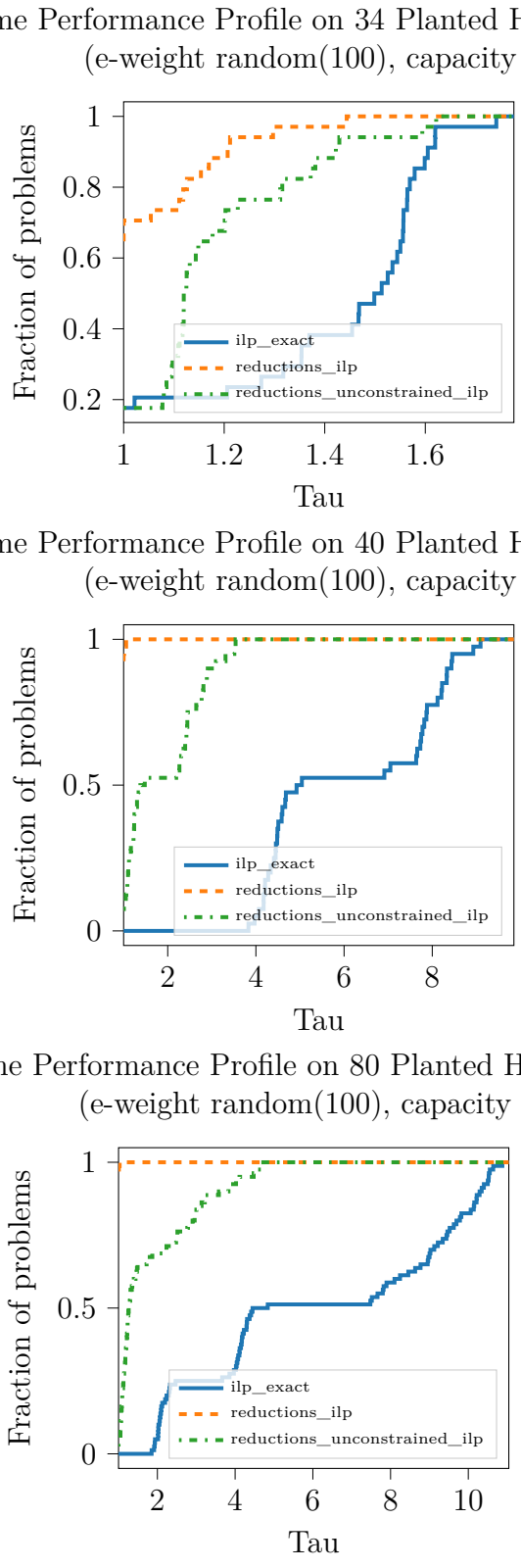
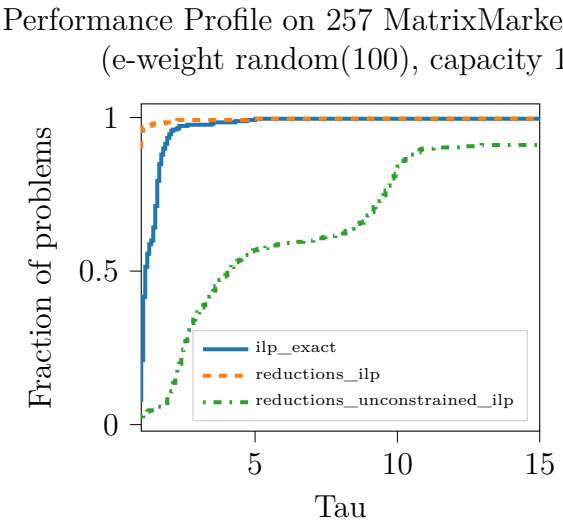
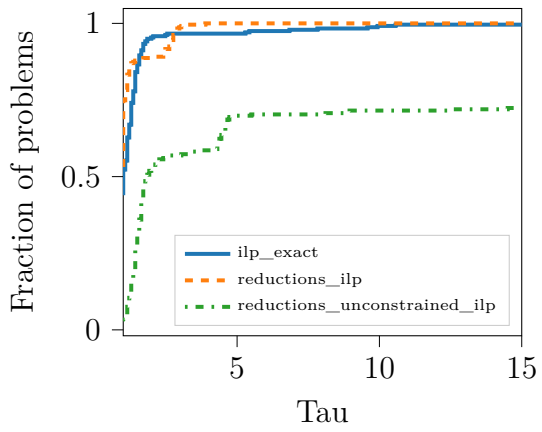


Figure 8.1: Impact of the reductions on exactly solvable planted hypergraphs shown in a time performance profile.

Constrained							Unconstrained						
# Vertices	# Edges	b(v)	Instances	# Edges	# Vertices	Speedup	# Vertices	# Edges	b(v)	Instances	# Edges	# Vertices	Speedup
24 000	4 000	1	10	3512.40	14390.70	1.04	24 000	4 000	1	10	3498.60	14367.60	0.98
		3	10	175.50	8957.40	7.79			3	10	172.00	8955.70	2.51
		5	10	0.60	8852.20	8.08			5	10	0.60	8852.20	1.94
	8 000	1	10	7508.20	14329.80	1.53		8 000	1	10	7491.90	14294.80	1.24
		3	10	1560.20	1065.80	4.31			3	10	1516.30	1040.50	3.36
		5	10	9.90	5.60	9.91			5	10	9.90	5.60	4.93
	12 000	5	10	2101.70	2360.60	4.24		12 000	5	10	2045.70	2338.80	3.60
	16 000	5	10	6755.20	3980.60	2.21		16 000	5	10	6716.70	3964.80	1.91
48 000	8 000	1	4	6998.50	28676.50	1.17	48 000	8 000	1	4	6962.50	28617.50	1.33
		3	10	344.90	17854.50	8.42			3	10	338.80	17851.90	3.50
		5	10	0.30	17650.20	9.16			5	10	0.30	17650.20	3.05
	16 000	1	10	15038.70	28718.60	1.55		16 000	1	10	15009.40	28661.80	1.37
		3	10	3039.80	2069.60	4.51			3	10	2964.60	2028.90	4.30
		5	10	18.30	10.40	10.44			5	10	18.30	10.40	7.70
	24 000	5	10	4346.10	4785.50	4.14		24 000	5	10	4219.70	4731.30	3.81
	32 000	5	10	13554.10	8014.90	2.03		32 000	5	10	13484.00	7984.80	1.83

Table 8.3: Impact of the constrained and unconstrained reductions on the exactly solvable instances' edge and vertex count for 6-uniform, 6-partite hypergraphs. The average vertex count includes the remaining vertices after reductions, while the edge count only contains unsolved edges. Speedup is relative to solving directly.





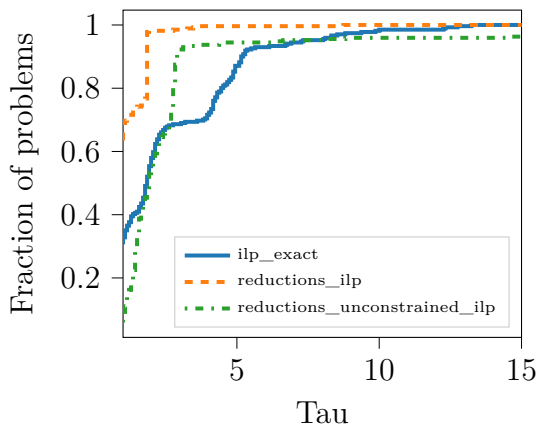
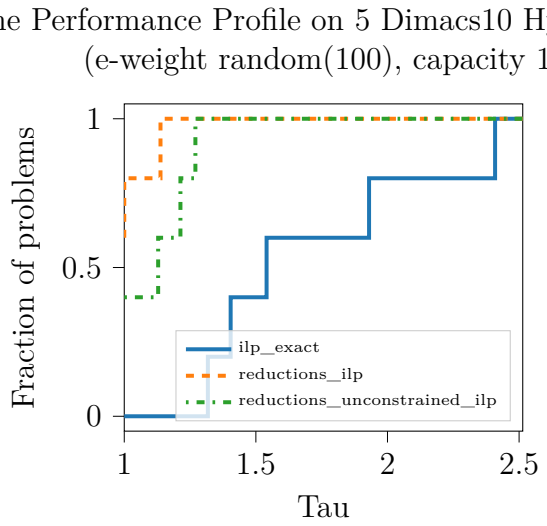
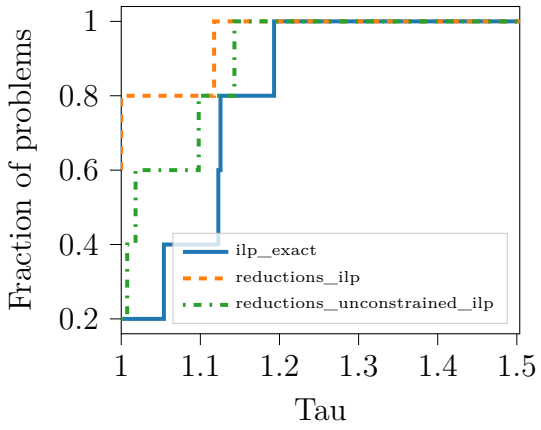


Figure 8.2: Impact of the reductions on exactly solvable matrix market incident hypergraphs displayed in a time performance profiles.





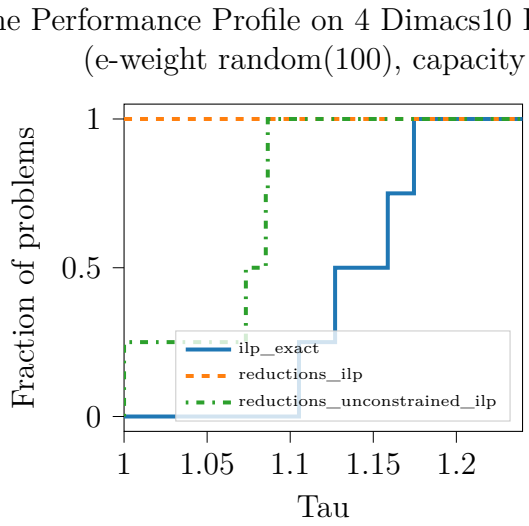
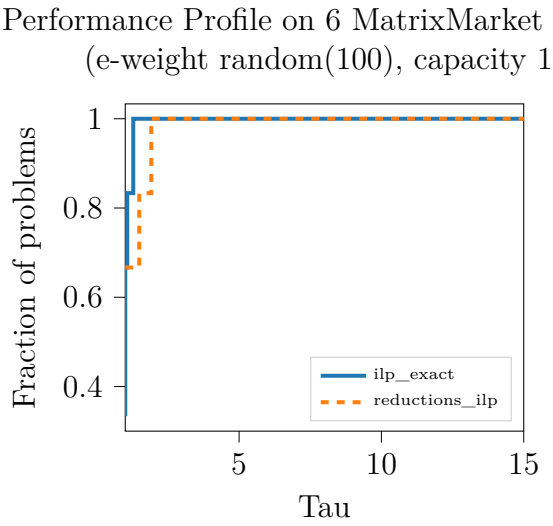
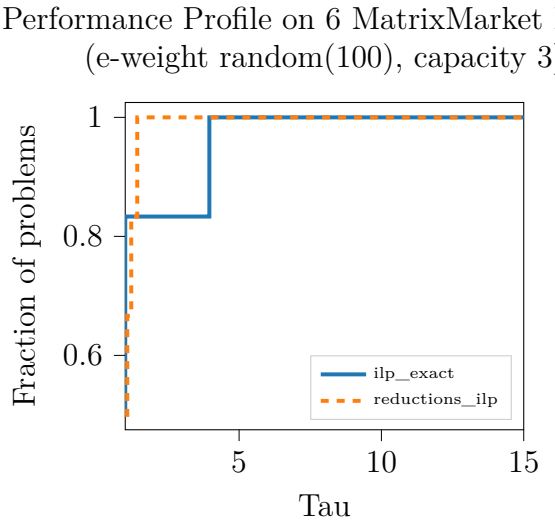


Figure 8.3: Impact of the reductions on exactly solvable dimacs10 hypergraphs shown in a time performance profile.





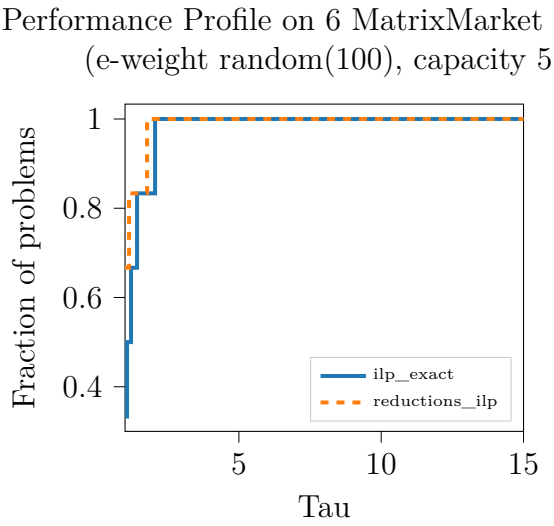


Figure 8.4: Impact of the reductions on exactly solvable graphs from the Florida Sparse Matrix Collection.

Algorithm Name	Type	Description
bmindegree_dynamic	dynamic greedy	descending weight scaled by residual capacity/degree
bratio_dynamic	dynamic greedy	descending weight scaled by residual capacity
bratio_static	greedy	descending weight scaled by capacity
bmult_static	greedy	descending weight scaled by minimum capacity of vertices
default_order	greedy	default order
bweight	greedy	descending weight

Table 8.4: Algorithm configuration for priority functions. Dynamic greedy approaches are recomputed after every insertion.

8.4 Comparing Priority Functions with bSuitor

In this experiment we are comparing our priority functions devised in Section 5 on RMAT graph instances and those graphs from the Florida Sparse Matrix Collection, selected by Khan et al. [48] with the bSuitor implementation provided by Khan et al. [48]. A summary of the configuration can be found in Table 8.4.

RMAT Instances. The performance profiles are shown in Figure 8.4. On the RMAT instances the performance of bSuitor is exactly matched by the priority function solely based on weight (bweight) on all capacities. The priority function based on the edge weight scaled by the residual capacity divided by the degree (bmindegree_dynamic) outperforms all other functions on capacity 1, but as the capacity per node grows, falls behind the weight based functions. For higher capacity the scaling by residual capacity per node (bratio_dynamic) works very well. The unordered adding of edges (default_order) trails all other functions.

The difference between the bmindegree_dynamic and bratio_dynamic is only the division by the node degree. As the capacity grows this scaling seems to be infeasible and degrade performance. The impact is so strong, that the performance is worse than the simple weight based functions. The improvement on this type of graph by scaling the weight with the residual capacity is about 1/20th and reasonable.

Florida Sparse Matrix Collection Instances. In Figure 8.4 the results on the 10 selected instances from the Florida Sparse Matrix Collection are shown. The matching problem ($b(v) = 1$) the same bmindegree_dynamic dominates the results and as capacity grows does not fall behind the weight based priority functions. Furthermore, the results match those on the RMAT instances: The bratio_dynamic, scaling

by the degree, works very well on higher capacity and on some instances yields a 10% improvement over the other approaches. In this experiment we solely looked at the result quality. The bSuitor algorithm by Khan et al. [48] is up to 20 times faster than our naive greedy implementation.

8.5 Local Search Experiments

In the following section we investigate the quality improvement by our iterated local search framework, devised in Section 6.1, and compare it to previous work by Dufosse et al. [20] on d–uniform, d–partite hypergraphs with uniform weight. The two approaches kss and ksmd by Dufosse et al. [20] were linked into our programs. For the Karp–Sipser scaling approach kss we chose 20 iterations. We compare these two approaches with our approaches, which combine a greedy approach with iterated local search and reductions.

6-**uniform,** 6-**partite Hypergraphs.** The results on planted hypergraphs with uniform weight are shown in Figure 8.5. The bratio_dynamic trails all other approaches due to the uniform weight and reaches on some instances only 50% of the size. The combination of reductions and iterated local search and iterated local search alone perform equally good and solve approximately 80% of the instances better than approaches by Dufosse et al. [20]. The ksmd solves roughly 25% on the same level as our approaches, reaches the level of our approaches at 0.9 and behaves exactly the same for lower τ. Some instances are solved better by the kss approach by Dufosse et al. [20] then the ksmd and our approaches, as seen by the fact, that the kss approach first reaches 100%.

8.6 Local Improvement Experiments

In this section we compare our iterated local search approach to the local improvement approach on different capacities. We compare the local improvement with range 100 and 1000 with the iterated local search, both get 15 or 100 seconds of computation time respectively and run ten repetitions.

6-**uniform,** 6-**partite Hypergraphs.** In Figure 8.6 the results for the iterated search vs local improvement on planted hypergraphs with capacity 1 and 3 are displayed. On capacity 1 the iterated local search for 100 seconds yields the best results. The time constrained version for 15 seconds returns on these hypergraphs second, while the local improvement strategies yield considerably worse results. In contrast, on capacity 3 the local improvement with 1000 edges for 100 seconds run time outperforms all other techniques.

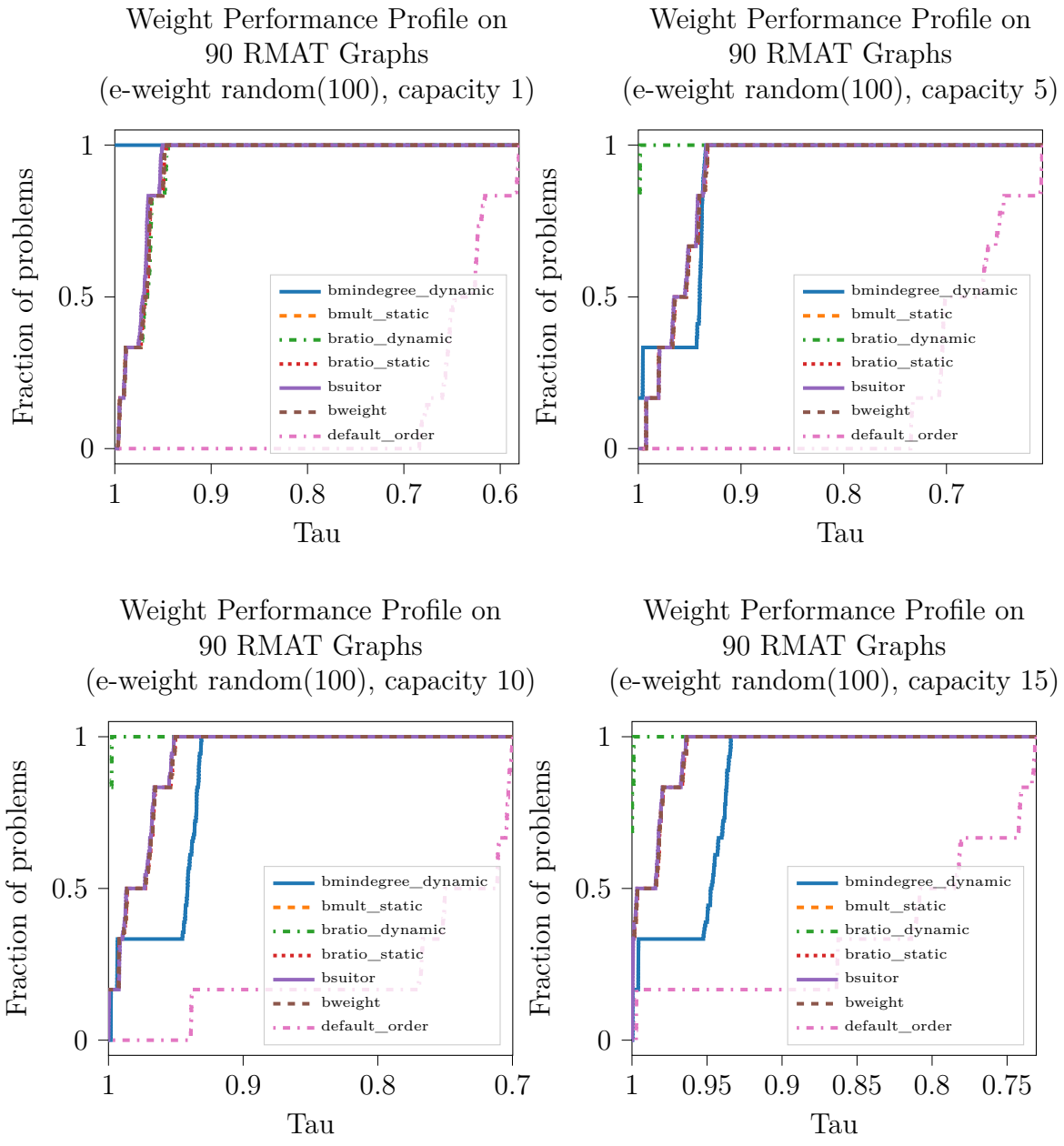


Figure 8.5: Performance Profile showing different priority functions on 90 RMAT instances with different capacities.

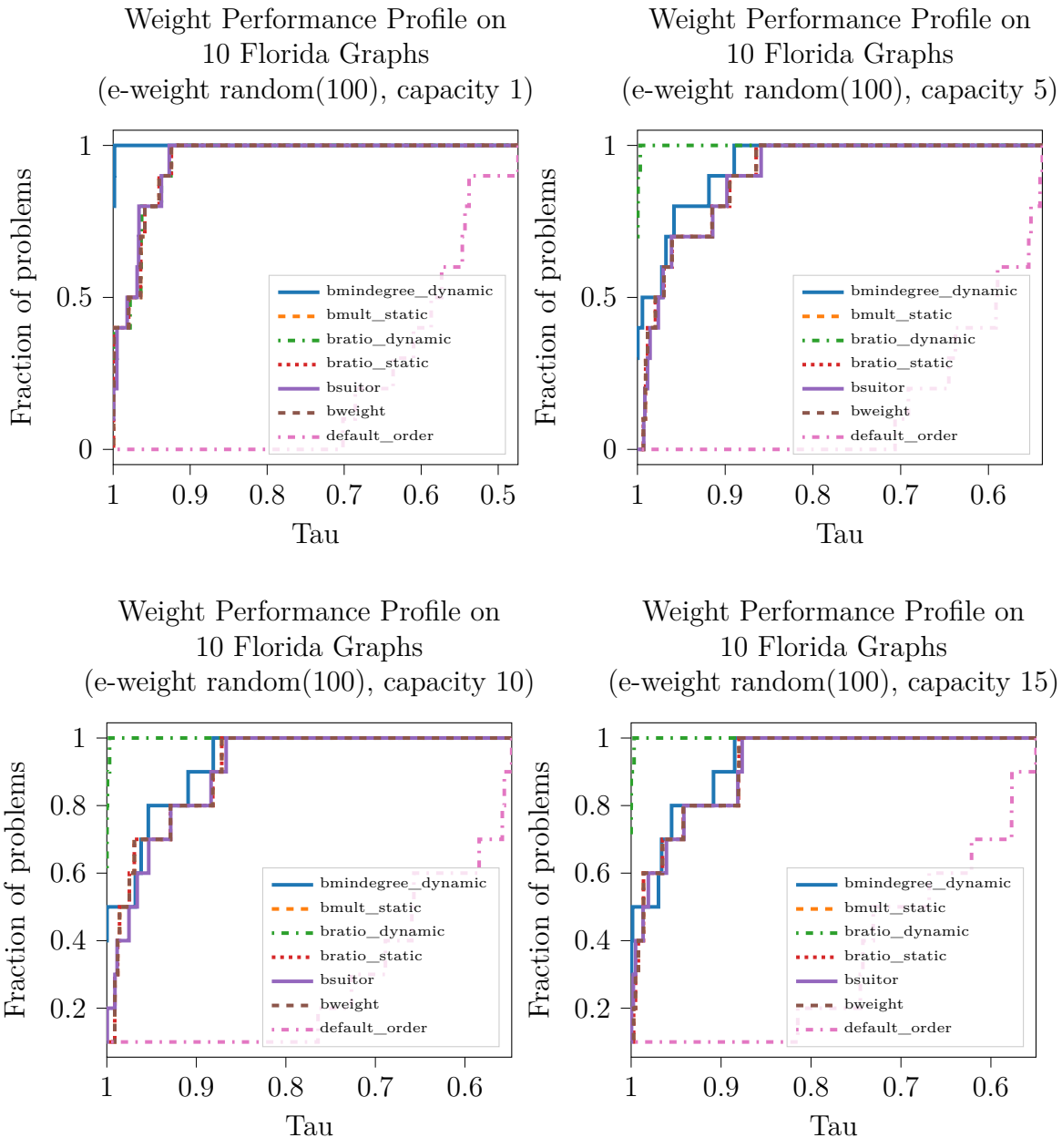


Figure 8.6: Performance Profile showing different priority functions on 10 Florida Sparse Matrix Collection instances with different capacities.

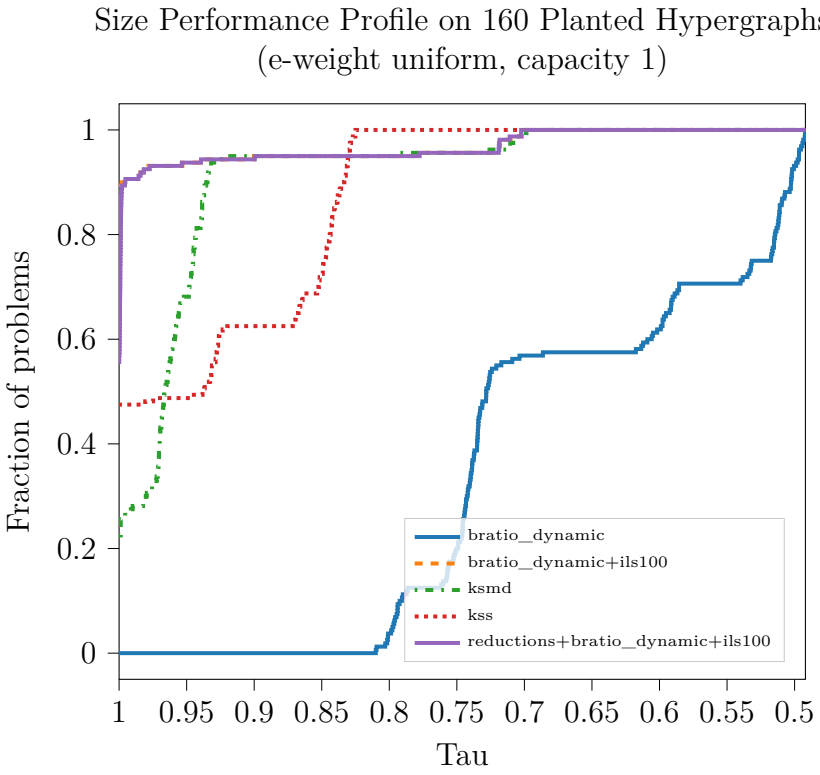


Figure 8.7: Performance Profile showing different approaches by Dufosse et al. [20] compared to our local search approach on 160 6–uniform, 6–partite hypergraph instances.

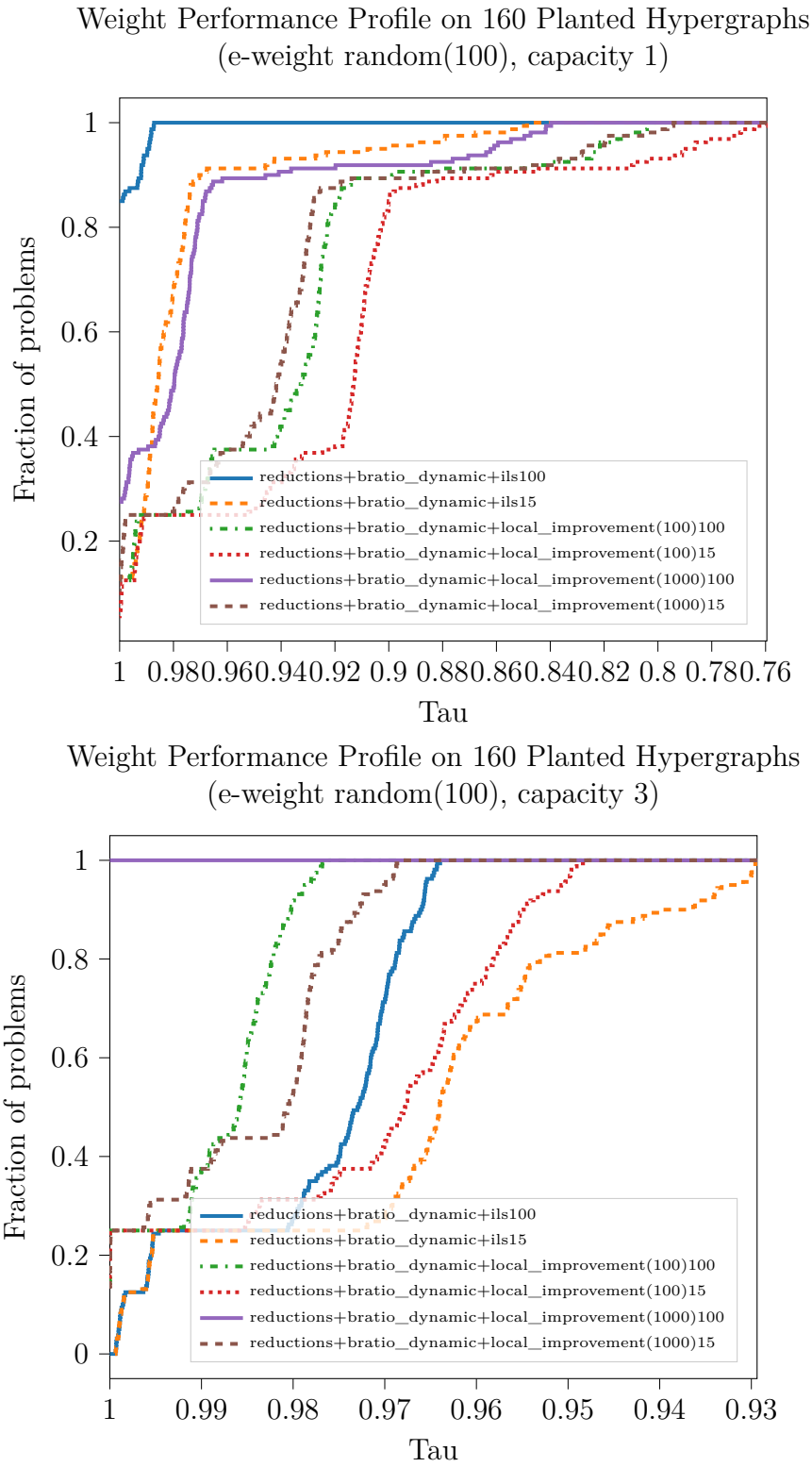


Figure 8.8: Performance Profile showing different iterated local search and local improvement strategies on planted ($d = 6$) instances.

Matrix Market Instances. The local improvement strategy failed due to too high memory consumption while solving the ILP with Gurobi on one of the instances. On the remaining 328 instances from the Matrix Market, the iterated local search for 100 seconds dominates the local improvement schemes for capacity 1. The local improvement scheme with higher distance return better results. The difference between the one that ran for 100 and 15 seconds is not big. With distance 100 it requires a $\tau = 0.75$ to solve all instances.

On capacity 3 the difference is not that pronounced (overall τ of 0.85), but the iterated local search trails all local improvement strategies. The longer the distance the better the results for the local improvement scheme are.

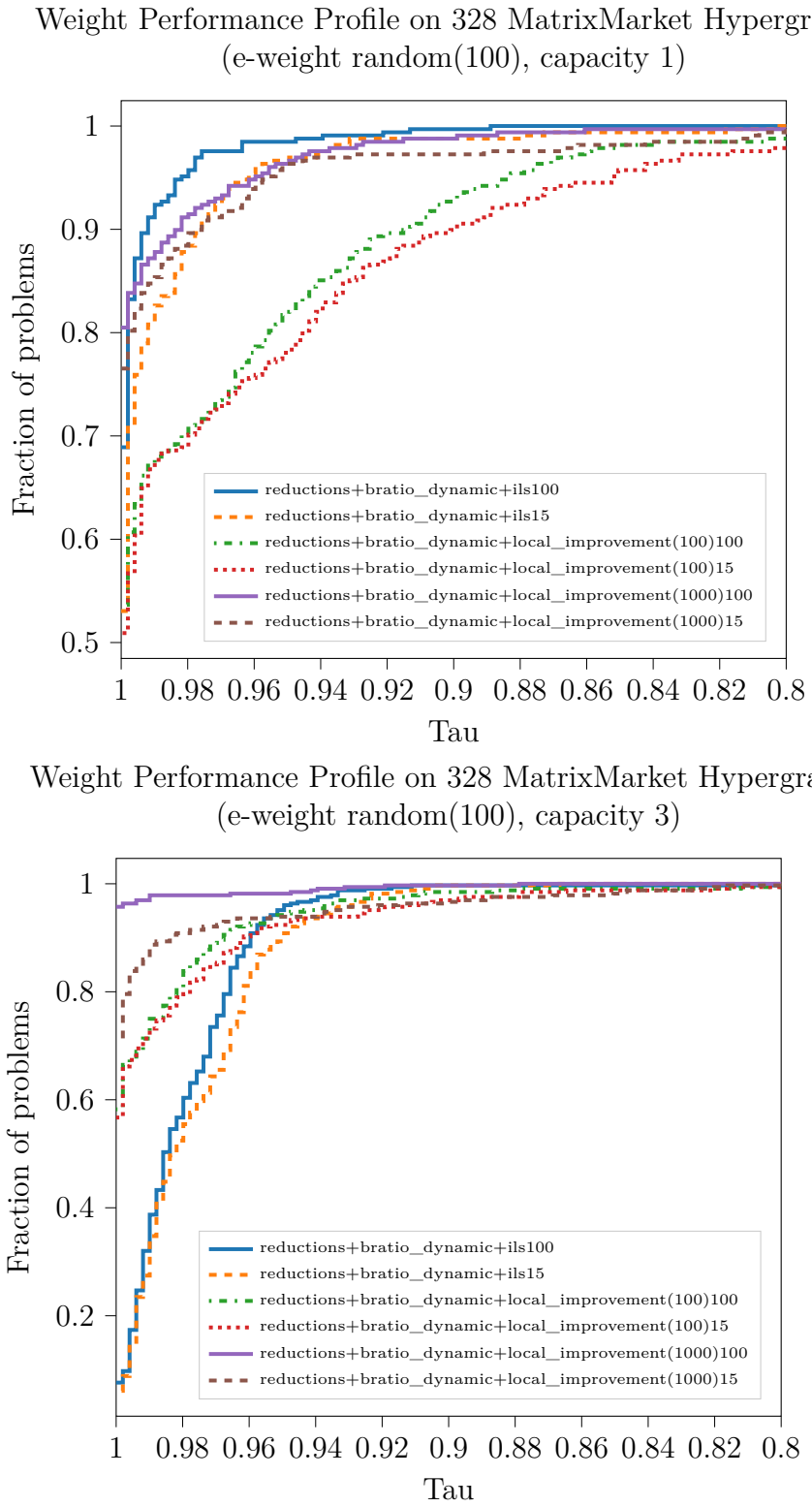


Figure 8.9: Performance Profile showing different iterated local search and local improvement strategies on matrix market instances.

CHAPTER 9

Discussion

In the following we discuss the results obtained from our experiments. We start with the exact reductions on hypergraphs, cover the results on graph b-matching, hypergraph matching and examine the results on our local improvement and iterated local search approach. Afterwards, we conclude and give an outlook over possible future work in the field.

Exact Reductions on Hypergraphs and Graphs. We engineered several data reductions for the problem such as the Neighborhood Removal and the Weighted Isolated Edge Removal identifying solution edges. The Weighted Twin and Edge Folding reductions alter the graph and postpone decisions, while the Weighted Domination and Abundant Vertices reduction exclude non-solution edges and prune vertices. The residual problem is solved as integer linear program exactly with Gurobi [38]. The experiments show that we can speed up a black box solver like Gurobi with our reductions in comparison to just solve them directly. The planted hypergraphs by Dufosse et al. [20] are more prone to our reductions than the other types of hypergraphs. On the planted hypergraphs, the higher the capacity the more pronounced the speedup is. However, for the other hypergraph classes the speedup is higher for capacity 1 than for the higher capacities. Nevertheless, we can report a significant speed up over all classes of hypergraphs. However, on a few instances it takes more time to search for the reductions than to simply solve the whole b-matching problem directly, even with the constrained version of our reductions. By restricting the search space and the number of passes we achieve a good balance between speed up and reducing the problem eagerly. For undirected graphs, like those selected by Khan et al. [48], the speedup in the current implementation is not that pronounced. Furthermore, the problem sizes are different from those of the incident hypergraphs which contain far fewer edges. In our experiments we used static numbers for the size restrictions, number of search iterations and all reductions at once, which is definitely worth further investigations.

Graph b-Matching Problem. We developed weight and (residual) capacity based heuristics for greedily computing initial solutions for the matching problem. These heuristics are dynamically, because they rely on updates after the insertion of an edge to the matching. In the case of graph b-matching, our priority approaches yield better results on the RMAT and matrix based instances than the simply weight based bSuitor greedy approaches, implemented by Khan et al. [48]. Nevertheless, our results confirm, that the implementation by Khan et al. [48] is very efficient and fast for obtaining approximating solutions. It might be worth considering to use similar techniques, like those bidding structures, to find a good initial matching in hypergraphs. These solutions could be then improved by either iterated local search or the local improvement scheme.

Hypergraph Matching Problem. The results from the initial computed solutions can be improved by the iterated local search that we developed in Chapter 6. On d-uniform, d-partite hypergraphs our iterated local search can boost the solution quality drastically and outperform the results of Karp–Sipser–Scaling by Dufosse et al. [20] except for a few instances for the simple cardinality matching problem in hypergraphs. Since the approaches by Dufosse et al. [20] only work on uniform weighted hypergraphs our reductions are not feasible on these instances. It might be worth considering using the Karp–Sipser–Scaling solution as base for our improvements via iterated local search on uniform edge weight instances, guaranteeing an equal or better result than the simple Karp–Sipser–Scaling for all instances alone.

Local Improvement and Iterated Local Search. In Chapter 4 we also developed an improvement scheme which solves sub graphs exactly using an integer linear program. This technique is very similar to the iterated local search, as both techniques try to find local improvements. Our iterated local search approach yields good results compared to the local improvement strategy via an ILP for small capacities. The higher the number of edges considered in the local improvement scheme is, the better solution quality we can achieve. In some instances of the matrix market this setting implied that we almost solved the whole or half hypergraph, which is unfeasible with data of real-world size. Additionally, we could not estimate the memory consumption of Gurobi [38] on those instances beforehand, whereas the memory consumption for the iterated local search is bounded.

The use of Gurobi [38] as black box solver for these sub problems and for exactly solving instances should be further discussed. The black box characteristic of this solver makes it hard to understand, how our reductions reduce the execution time. Furthermore, the memory consumption of Gurobi can not be derived only by the size of the problem and is sometimes very high, causing the program to fail and requiring a rerun with fewer concurrent processes.

9.1 Conclusion

In this thesis we presented six novel exact reduction for the b-matching problem in hypergraphs, heuristics for computing a good initial matching and strategies to improve the quality of the matching by either iterated local search or a local improvement scheme. Furthermore, we presented an efficient data structure to store b-matching and modifiable hypergraphs, as some of our reductions change the shape of our hypergraphs. In experiments we showed the competitiveness of these approaches in comparison with recent results by Dufosse et al. [20] and Khan et al. [48].

The first two reductions, Neighborhood Removal and Weighted Isolated Edge Removal, identify edges with high weight and dominating their neighborhood weight wise, allowing us to determine that they are part of the solution. The Weighted Edge Folding reduction combines three or more edges to an edge and postpones the decision to a later point. The Weighted Twin reduction searches for edges that have common neighbors and either applies a variant of the Neighborhood Removal or the Weighted Edge Folding reduction, depending on the weight configuration of the common neighborhood. The Weighted Domination reduction finds edges that are guaranteed to be not part of the solution, as they are weight dominated by an edge formed by a subset of vertices. Our last reduction, the Abundant Vertices Reduction removes all vertices, that do not constitute a decision problem and adds vanishing edges to the solution.

The initial solutions we have obtained from weight heuristics can be improved by an iterated local search or a local improvement strategy. The iterated local search works by identifying edges that can be swapped for a solution edge, increasing the weight of the b-matching leading towards a local optima. By applying perturbation we can escape those local optima. The local improvement scheme solves sub problems exactly and improves the solution quality.

We presented two data structures for modifiable hypergraphs and b-matching. Our timestamping framework for edges allows us to quickly skip edges and vertices that have not been changed since the last scan.

We implemented our reductions, iterated local search and local improvement strategy in C++ and thoroughly tested them in our experiments. The experiments show that all classes of hypergraphs benefit from the reductions, when they are solved exactly with Gurobi [38]. The static size constrains for the several reductions balance the speedup and reduction impact. Our initial solutions for the graph b-matching problem outperform those by Khan et al. [48]. For the matching problem in hypergraphs we can report that our iterated local search outperforms the Karp–Sipser–Scaling approaches by Dufosse et al. [20]. The iterated local search is very effective on low b values, as finding swaps is easier. For higher capacities finding swaps via local improvement strategies is more viable.

9.2 Future Work

In this thesis we focused on the development of novel reductions for the hypergraph b-matching problem. It would be interesting to look into exactly solving the problem using a branch and reduce framework utilizing our novel reductions. Furthermore, we restricted our reductions by forcing neighboring edges to be adjacent via degree-2 vertices, it might be worth investigating, how we could relax the conditions of our reductions and broaden their applicability. Additionally, we could investigate, how ordering the entries for edges and vertices in memory could help reduce algorithmic complexity, when searching for subsets or merging edges.

Furthermore, the rise of parallel computing architectures calls for the adaptation of parallelism in our reductions search and iterated local search framework. Examples for good parallel implementation ideas for finding an initial matching for graphs are shown in the work of Khan et al. [48], maybe it is possible to transfer those to the b-matching problem in hypergraphs.

On the modelling side, the weight function could be made submodular to have a similar problem to those described by Ferdous et al. [25] for normal graph b-matching. Finally, it might be interesting to turn our attention to dynamic or online hypergraph b-matching problems. In this problem the hypergraph is not known in advance or gets modified after during the process by edge insertions and deletions. The matching needs to be updated after every update in the hypergraph and keeping quality guarantees would be very challenging.

Zusammenfassung

Ein Hypergraph ist eine Generalisierung eines Graphens, in dem mehr als zwei Knoten in einer Kante sein können. Für das gewichtete b-Matching Problem gilt es die höchste Summe von gewichteten Kanten auszuwählen, während für jeden Knoten eine Maximalanzahl an Kanten ausgewählt sein darf. In dieser Arbeit präsentieren wir sechs neue, exakte Reduktionen für dieses Problem und eine iterierte lokale Suche, sowie ein lokales Verbesserungsschemata. Die exakten Reduktionen erlauben uns zu entscheiden, ob Kanten im Matching enthalten sind oder gar nicht Teil der Lösung sein können. Die iterierte lokale Suche funktioniert über das Finden von möglichen verbesserenden Wechseln von zwei Kanten für eine Lösungskante, während das lokale Verbesserungsschemata einen Teilgraphen exakt löst. In Experimenten zeigen wir die Effektivität unserer Reduktionen und das Potential der iterierten lokalen Suche und des lokalen Verbesserungsschemata zur Verbesserung von Lösungen, die wir durch einfache gewichtsbasierte Heuristiken erhalten.

Bibliography

[1] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast local search for the maximum independent set problem. *J. Heuristics*, 18 (4):525–547, 2012. doi: 10.1007/s10732-012-9196-4. URL `https://doi.org/10.1007/s10732-012-9196-4`.

[2] Georg Anegg, Haris Angelidakis, and Rico Zenklusen. *Simpler and Stronger Approaches for Non-Uniform Hypergraph Matching and the Füredi, Kahn, and Seymour Conjecture*, pages 196–203. doi: 10.1137/1.9781611976472.22. URL `https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.22`.

[3] Eugenio Angriman, Henning Meyerhenke, Christian Schulz, and Bora Uçar. Fully-dynamic weighted matching approximation in practice. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*, pages 32–44. SIAM, 2021.

[4] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. *Algorithm engineering: Selected results and surveys*, pages 19–80, 2016.

[6] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, volume 8097 of *Lecture Notes in Computer Science*, pages 659–670. Springer, 2013. doi: 10.1007/978-3-642-40047-6_66. URL `https://doi.org/10.1007/978-3-642-40047-6_66`.

[7] Ulrik Brandes, Linton C. Freeman, and Dorothea Wagner. Social networks. In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization*, pages 805–839. Chapman and Hall/CRC, 2013.

[8] Shaowei Cai, Wenying Hou, Jinkun Lin, and Yuanjie Li. Improving local search for minimum weight vertex cover by dynamic strategies. In *IJCAI*, pages 1412–1418, 2018.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[10] Krzysztof M Choromanski, Tony Jebara, and Kui Tang. Adaptive anonymity via b-matching. *Advances in Neural Information Processing Systems*, 26, 2013.

[11] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[12] Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 509–518. IEEE, 2013.

[13] Marek Cygan, Fabrizio Grandoni, and Monaldo Mastrolilli. How to sell hyperedges: The hypermatching assignment problem. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 342–351. SIAM, 2013.

[14] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Accelerating local search for the maximum independent set problem. In *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings 15*, pages 118–133. Springer, 2016.

[15] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL `https://doi.org/10.1145/2049662.2049663`.

[16] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi: 10.1007/s101070100263. URL `https://doi.org/10.1007/s101070100263`.

[17] Yuanyuan Dong, Andrew V Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio GC Resende, and Quico Spaen. A local search algorithm for large maximum weight independent set problems. In *30th Annual European Symposium on Algorithms (ESA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[18] Doratha E Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[19] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1), jan 2014. ISSN 0004-5411. doi: 10.1145/2529989. URL `https://doi.org/10.1145/2529989`.

[20] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Effective heuristics for matchings in hypergraphs. In *International Symposium on Experimental Algorithms*, pages 248–264. Springer, 2019.

[21] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17: 449–467, 1965. doi: 10.4153/CJM-1965-045-4.

[22] Jack Edmonds and Ellis L Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5:88–124, 1973.

[23] Mourad El Ouali and Gerold Jäger. The b-matching problem in hypergraphs: Hardness and approximability. In Guohui Lin, editor, *Combinatorial Optimization and Applications*, pages 200–211, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31770-5.

[24] Mourad El Ouali, Antje Fretwurst, and Anand Srivastav. Inapproximability of b-matching in k-uniform hypergraphs. In Naoki Katoh and Amit Kumar, editors, *WALCOM: Algorithms and Computation*, pages 57–69, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19094-0.

[25] S M Ferdous, Alex Pothen, Arif Khan, Ajay Panyala, and Mahantesh Halappanavar. *A Parallel Approximation Algorithm for Maximizing Submodular b-Matching*, pages 45–56. doi: 10.1137/1.9781611976830.5. URL `https://epubs.siam.org/doi/abs/10.1137/1.9781611976830.5`.

[26] Fedor V Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM (JACM)*, 56 (5):1–32, 2009.

[27] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[28] Martin Fürer and Huiwen Yu. Approximating the k-set packing problem by local improvements. In *Combinatorial Optimization - Third International Symposium, ISCO 2014, Revised Selected Papers*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 408–420, Germany, 2014. Springer Verlag. ISBN 9783319091730. doi: 10.1007/978-3-319-09174-7_35. 3rd International Symposium on Combinatorial Optimization, ISCO 2014 ; Conference date: 05-03-2014 Through 07-03-2014.

[29] Harold N Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 448–456, 1983.

[30] Harold N Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 434–443, 1990.

[31] Harold N Gabow and Robert E Tarjan. Faster scaling algorithms for general graph matching problems. *Journal of the ACM (JACM)*, 38(4):815–853, 1991.

[32] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[33] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 90–100. SIAM, 2008.

[34] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 128–142. SIAM, 2021.

[35] Giorgos Georgiadis and Marina Papatriantafilou. Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists. *Algorithms*, 6(4):824–856, 2013.

[36] Martin Grötschel and Olaf Holland. Solving matching problems with linear programming. *Mathematical Programming*, 33:243–259, 1985.

[37] Jiewei Gu, Weiguo Zheng, Yuzheng Cai, and Peng Peng. Towards computing a near-maximum weighted independent set on massive graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 467–477, 2021.

[38] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL `https://www.gurobi.com`.

[39] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms, 2021. URL `https://arxiv.org/abs/2102.11169`.

[40] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *computational complexity*, 15(1):20–39, 2006. doi: 10.1007/s00037-006-0205-6. URL `https://doi.org/10.1007/s00037-006-0205-6`.

[41] Rolf H Höhring, Matthias Müller-Hannemann, and Karsten Wiehe. Mesh refinement via bidirected flows: Modeling, complexity, and computational results. *Journal of the ACM (JACM)*, 44(3):395–426, 1997.

[42] Bert Huang and Tony Jebara. Fast b-matching via sufficient selection belief propagation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 361–369. JMLR Workshop and Conference Proceedings, 2011.

[43] Lorenz Hübschle-Schneider and Peter Sanders. Linear work generation of R-MAT graphs. *Network Science*, 8(4):543 – 550, 2020.

[44] Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105 – 142, 1999. doi: 10.1007/BF02392825. URL `https://doi.org/10.1007/BF02392825`.

[45] Tony Jebara, Jun Wang, and Shih-Fu Chang. Graph construction and b-matching for semi-supervised learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 441–448, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553432. URL `https://doi.org/10.1145/1553374.1553432`.

[46] Leonid V Kantorovich. On the translocation of masses. In *Dokl. Akad. Nauk. USSR (NS)*, volume 37, pages 199–201, 1942.

[47] Richard M Karp and Michael Sipser. Maximum matching in sparse random graphs. In *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, pages 364–375. IEEE, 1981.

[48] Arif Khan, Alex Pothen, Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM Journal on Scientific Computing*, 38(5):S593–S619, 2016. doi: 10.1137/15M1026304. URL `https://doi.org/10.1137/15M1026304`.

[49] Christos Koufogiannakis and Neal E Young. Distributed fractional packing and maximum weighted b-matching via tail-recursive duality. In *Distributed Computing: 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings 23*, pages 221–238. Springer, 2009.

[50] Piotr Krysta. Greedy approximation via duality for packing, combinatorial auctions and routing. In *Mathematical Foundations of Computer Science 2005: 30th International Symposium, MFCS 2005, Gdansk, Poland, August 29–September 2, 2005. Proceedings 30*, pages 615–627. Springer, 2005.

[51] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 144–158. SIAM, 2019.

[52] Kenneth Langedal, Johannes Langguth, Fredrik Manne, and Daniel Thilo Schroeder. Efficient minimum weight vertex cover heuristics using graph neural networks. In *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[53] Eugene L Lawler. Cutsets and partitions of hypergraphs. *Networks*, 3(3):275–285, 1973.

[54] Ruizhi Li, Shuli Hu, Haochen Zhang, and Minghao Yin. An efficient local search framework for the minimum weighted vertex cover problem. *Information Sciences*, 372:428–445, 2016.

[55] Hui Lin and Jeff Bilmes. Word alignment via submodular maximization over matroids. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 170–175, 2011.

[56] Rodica Ioana Lung, Noémi Gaskó, and Mihai Alexandru Suciu. A hypergraph model for representing scientific output. *Scientometrics*, 117(3):1361–1379, 2018. doi: 10.1007/s11192-018-2908-2. URL `https://doi.org/10.1007/s11192-018-2908-2`.

[57] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE, 2014.

[58] ALFRED BURTON MARSH III. *Matching algorithms*. The Johns Hopkins University, 1979.

[59] Franco Mascia, Elisa Cilia, Mauro Brunato, and Andrea Passerini. Predicting structural and functional sites in proteins by searching for maximum-weight cliques. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 1274–1279, 2010.

[60] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *WEA*, volume 7, pages 242–255. Springer, 2007.

[61] Julián Mestre. Greedy in approximation algorithms. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*, pages 528–539. Springer, 2006.

[62] Matthias Müller-Hannemann and Alexander Schwartz. Implementing weighted b-matching algorithms: insights from a computational study. *Journal of Experimental Algorithmics (JEA)*, 5:8–es, 2000.

[63] Bruno Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, 12(3):567–583, 2018. doi: 10.1007/s11590-017-1128-7. URL `https://doi.org/10.1007/s11590-017-1128-7`.

[64] Manfred W Padberg and M Ram Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982.

[65] Ojas Parekh and David Pritchard. Generalized hypergraph matching via iterated packing and local ratio. In *Approximation and Online Algorithms: 12th International Workshop, WAOA 2014, Wrocław, Poland, September 11-12, 2014, Revised Selected Papers*, pages 207–223. Springer, 2015.

[66] Marco Pavone, Amin Saberi, Maximilian Schiffer, and Matt Wu Tsao. Online hypergraph matching with delays. *Operations Research*, 70(4):2194–2212, 2022.

[67] Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS 99: 16th Annual Symposium on Theoretical Aspects of Computer Science Trier, Germany, March 4–6, 1999 Proceedings 16*, pages 259–269. Springer, 1999.

[68] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM J. Exp. Algorithmics*, mar 2022. doi: 10.1145/3529090. URL `https://doi.org/10.1145/3529090`.

[69] TW Strijk, AM Verweij, KI Aardal, et al. Algorithms for maximum independent set applied to map labelling. 2000.

[70] Robert L Thorndike. The problem of classification of personnel. *Psychometrika*, 15(3):215–235, 1950.

[71] Jeffrey S Warren and Illya V Hicks. Combinatorial branch-and-bound for the maximum weight independent set problem. *Relatório Técnico, Texas A&M University, Citeseer*, 9:17, 2006.

[72] Mingyu Xiao, Sen Huang, Yi Zhou, and Bolin Ding. Efficient reductions and a fast algorithm of maximum weighted independent set. In *Proceedings of the Web Conference 2021*, WWW '21, pages 3930–3940, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383127. doi: 10.1145/3442381.3450130. URL `https://doi.org/10.1145/3442381.3450130`.

[73] Ron Zass and Amnon Shashua. Probabilistic graph and hypergraph matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008. doi: 10.1109/CVPR.2008.4587500.